Partial State in Dataflow-Based Materialized Views

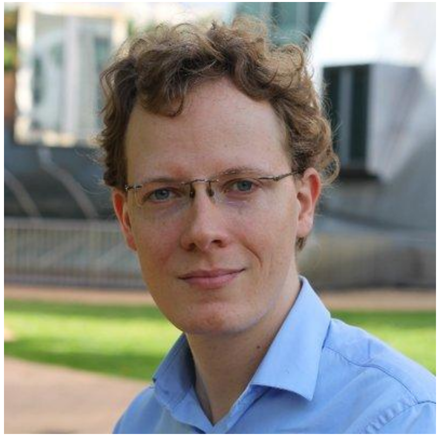
Jon Gjengset — Doctoral Dissertation
jon@thesquareplanet.com / @jonhoo

My Committee

Robert Morris
(thesis advisor)

M. Frans Kaashoek

Sam Madden



Malte Schwarzkopf

Why are we here?

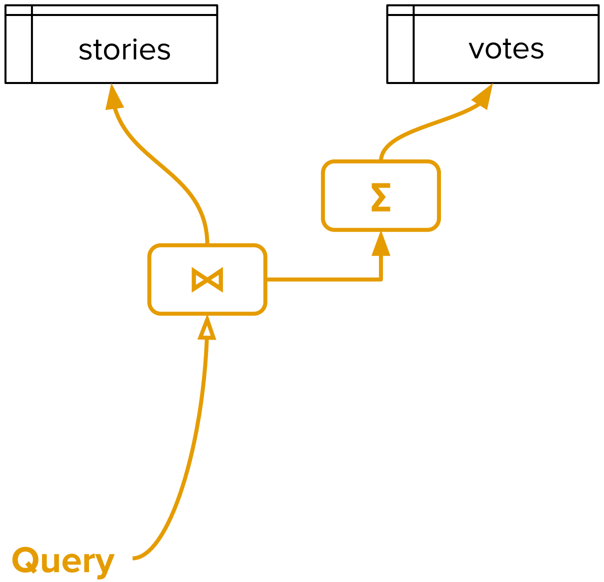
To make databases better.

Database 101

You take some tables.

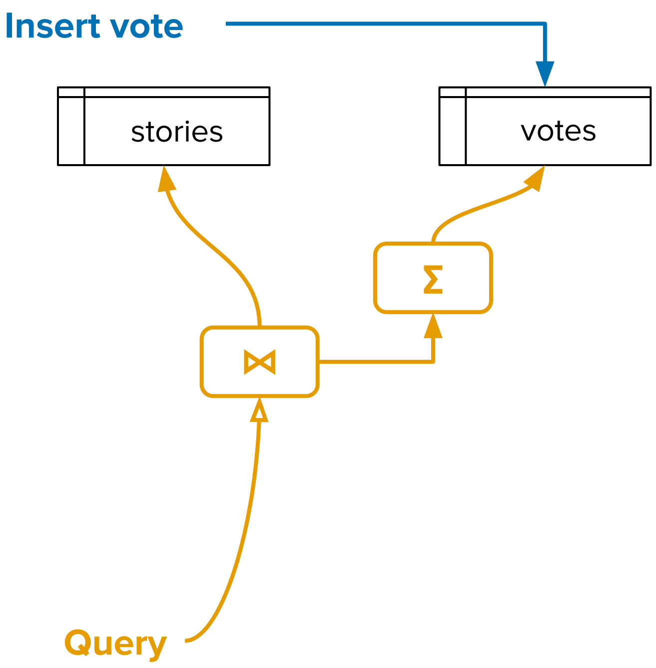
Database 101

To query, do this:



Database 101

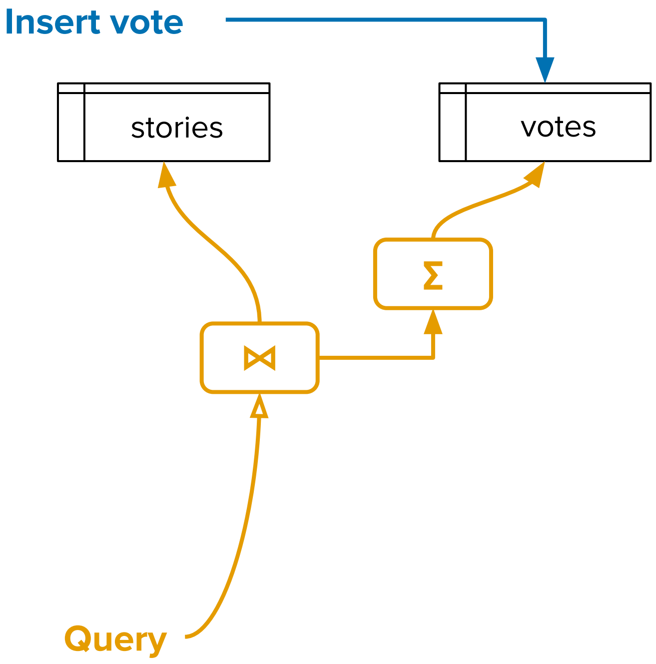
To update, do this:



Why are we here?

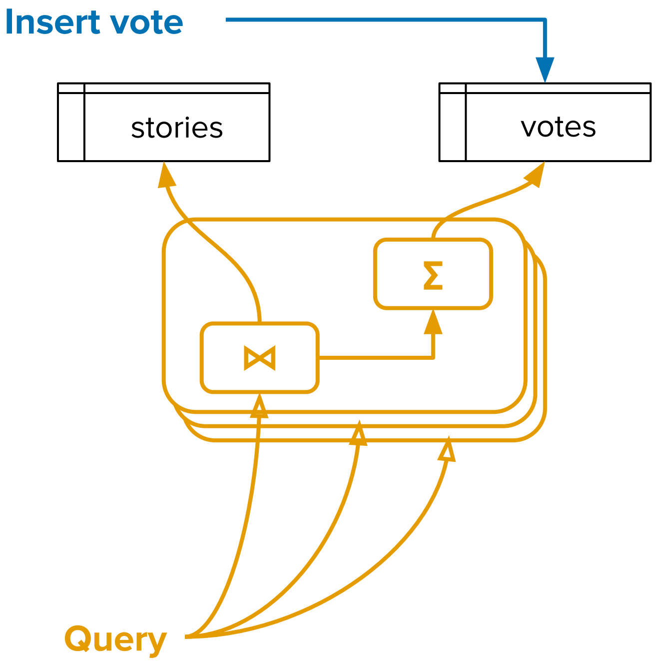
More orange work than blue.

But orange is often more common!



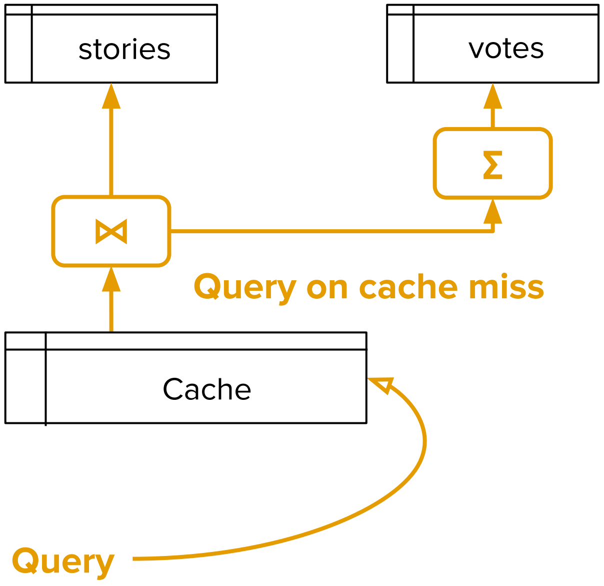
Why are we here?

Repeated, unnecessary orange work.



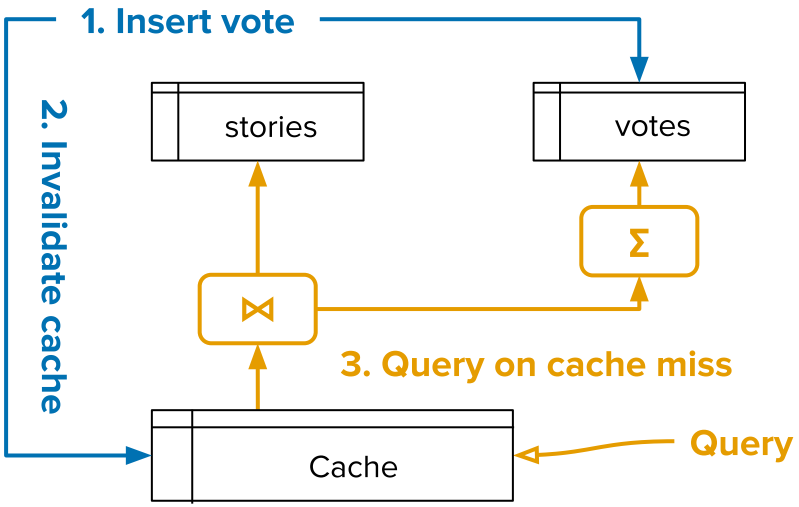
But Jon, caches.

Queries are now fast again!



Caches are great.

But caching is hard.



4. Fill in the cache..?

5. Evict from the cache..?

Automatic database caching.

Back to the title:

Partial State in Dataflow-Based Materialized Views

Back to the title:

Partial State in Dataflow-Based **Materialized Views**

Remembering Query Results

- Invented by the database community in the 1980s.
- Essentially “run the query and remember the result”.
- Key question is how to **maintain** the materialization:
  - What happens if the underlying data **changes**?
  - Should be **incremental**: don’t execute from scratch each time.
  - Maintain on **write** or on subsequent **read**?

Back to the title:

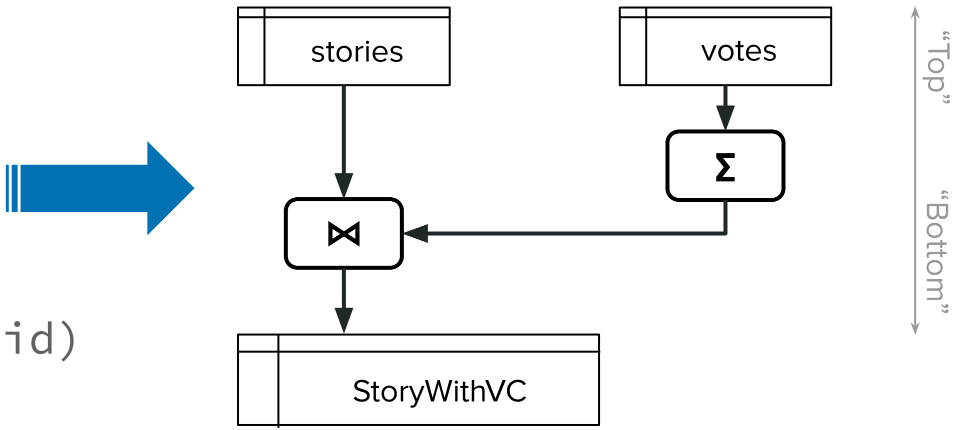
Partial State in **Dataflow-Based** Materialized Views

Push Changes to Views

- Dataflow has many definitions; here: data moves to compute.
  - Think "push-based computation".
- Data changes propagate through graph of *operators*.
  - Here: relational operators like joins, aggregations, and filters.
- Each edge is a data dependency.
  - e.g., a join depends on its inputs.
- Messages are *deltas*:
  - Each delta is a full row with a positive (add) or negative (remove) sign.

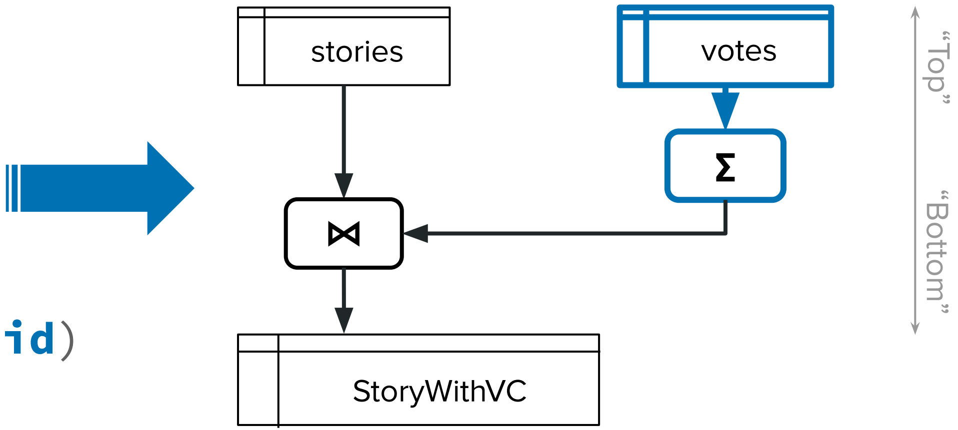
Example Dataflow Execution

```
CREATE MATERIALIZED VIEW
 StoryWithVC
AS SELECT
 stories.*,
 COUNT(votes.user) AS votes
FROM stories
JOIN votes
 ON (votes.story_id = stories.id)
GROUP BY stories.id;
```



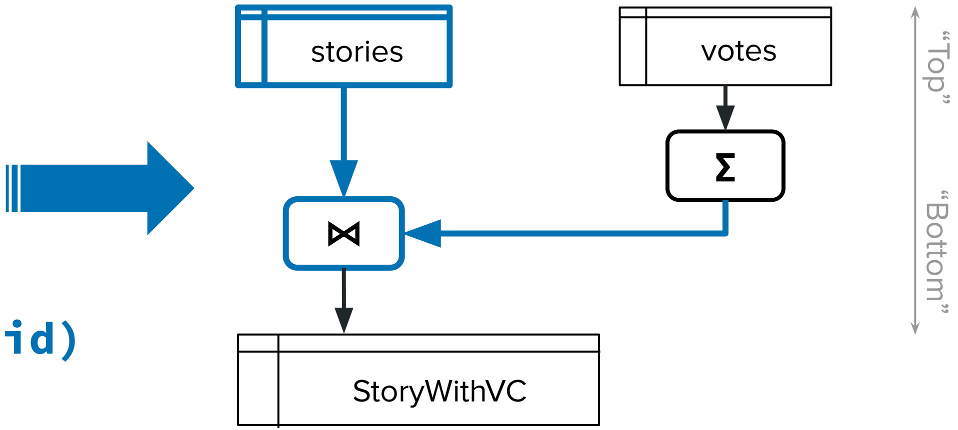
Example Dataflow Execution

```
CREATE MATERIALIZED VIEW
 StoryWithVC
AS SELECT
 stories.*,
 COUNT(votes.user) AS votes
FROM stories
JOIN votes
 ON (votes.story_id = stories.id)
GROUP BY stories.id;
```



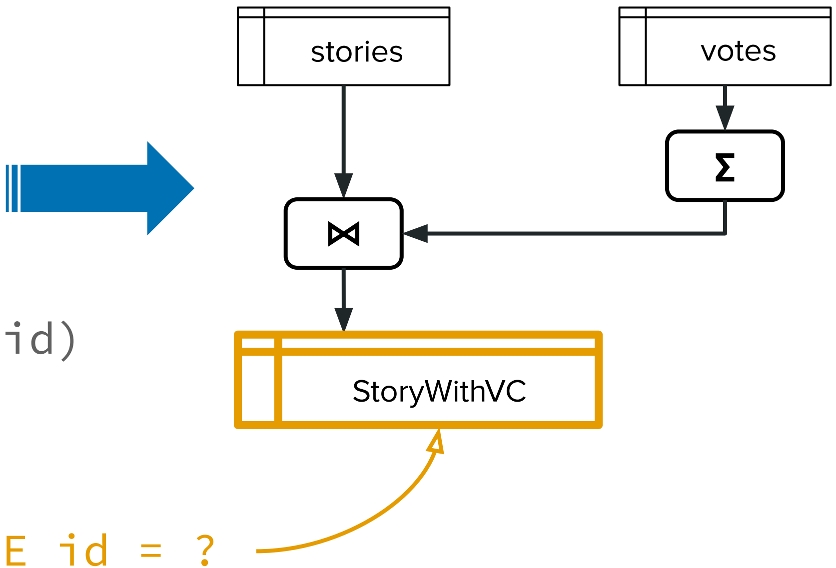
Example Dataflow Execution

```
CREATE MATERIALIZED VIEW
 StoryWithVC
AS SELECT
 stories.*,
 COUNT(votes.user) AS votes
FROM stories
JOIN votes
 ON (votes.story_id = stories.id)
GROUP BY stories.id;
```



Example Dataflow Execution

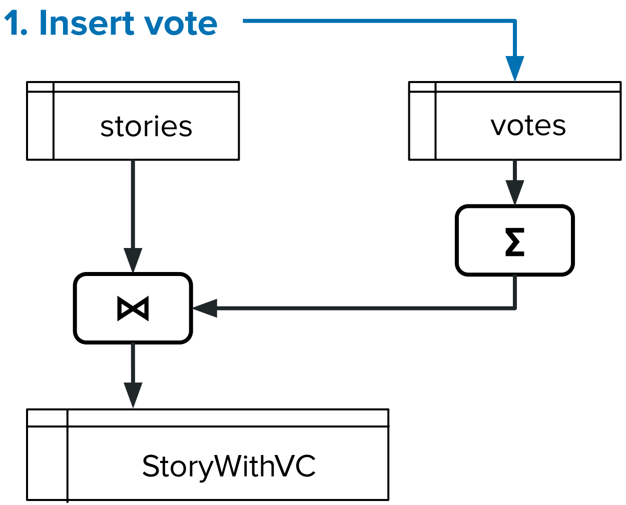
```
CREATE MATERIALIZED VIEW
 StoryWithVC
AS SELECT
 stories.*,
 COUNT(votes.user) AS votes
FROM stories
JOIN votes
 ON (votes.story_id = stories.id)
GROUP BY stories.id;
```




```
SELECT * FROM StoryWithVC WHERE id = ?
```

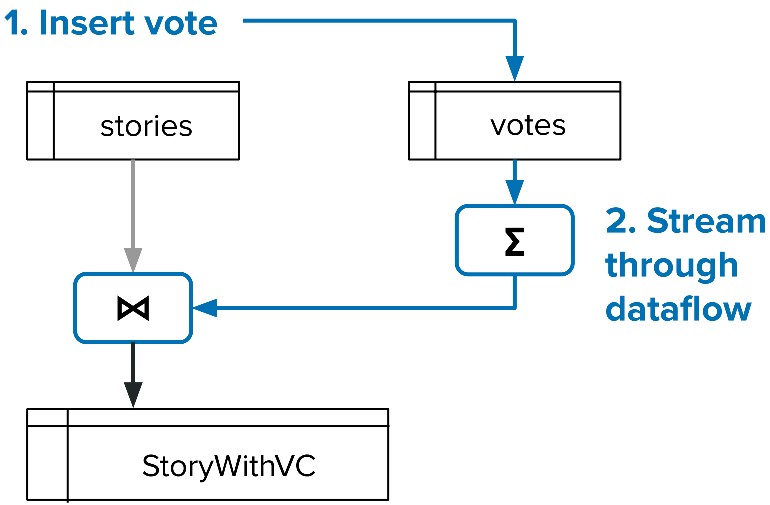
Example Dataflow Execution

```
CREATE MATERIALIZED VIEW
 StoryWithVC
AS SELECT
 stories.*,
 COUNT(votes.user) AS votes
FROM stories
JOIN votes
 ON (votes.story_id = stories.id)
GROUP BY stories.id;
```



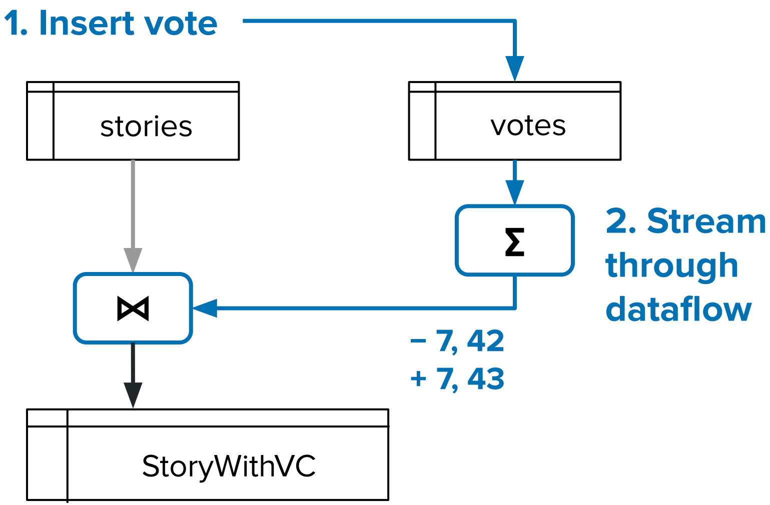
Example Dataflow Execution

```
CREATE MATERIALIZED VIEW
 StoryWithVC
AS SELECT
 stories.*,
 COUNT(votes.user) AS votes
FROM stories
JOIN votes
 ON (votes.story_id = stories.id)
GROUP BY stories.id;
```



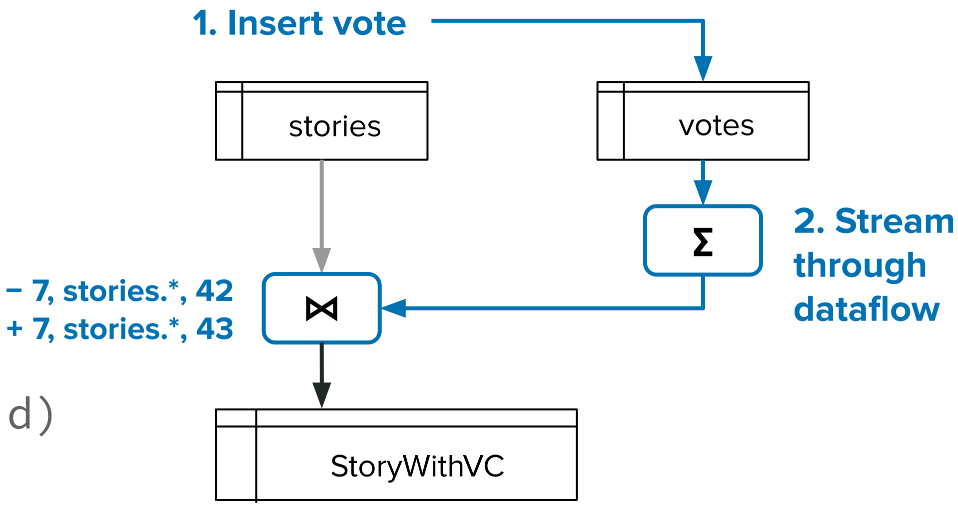
Example Dataflow Execution

```
CREATE MATERIALIZED VIEW
 StoryWithVC
AS SELECT
 stories.*,
 COUNT(votes.user) AS votes
FROM stories
JOIN votes
 ON (votes.story_id = stories.id)
GROUP BY stories.id;
```



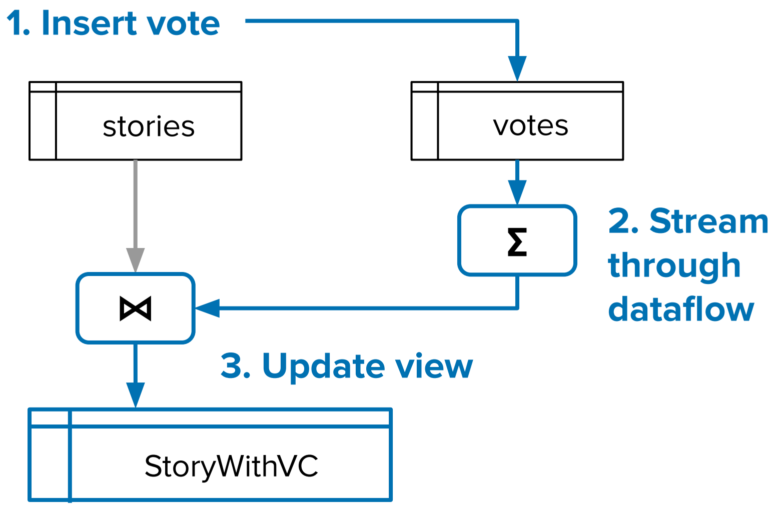
Example Dataflow Execution

```
CREATE MATERIALIZED VIEW
 StoryWithVC
AS SELECT
 stories.*,
 COUNT(votes.user) AS votes
FROM stories
JOIN votes
 ON (votes.story_id = stories.id)
GROUP BY stories.id;
```



Example Dataflow Execution

```
CREATE MATERIALIZED VIEW
 StoryWithVC
AS SELECT
 stories.*,
 COUNT(votes.user) AS votes
FROM stories
JOIN votes
 ON (votes.story_id = stories.id)
GROUP BY stories.id;
```



Back to the title:

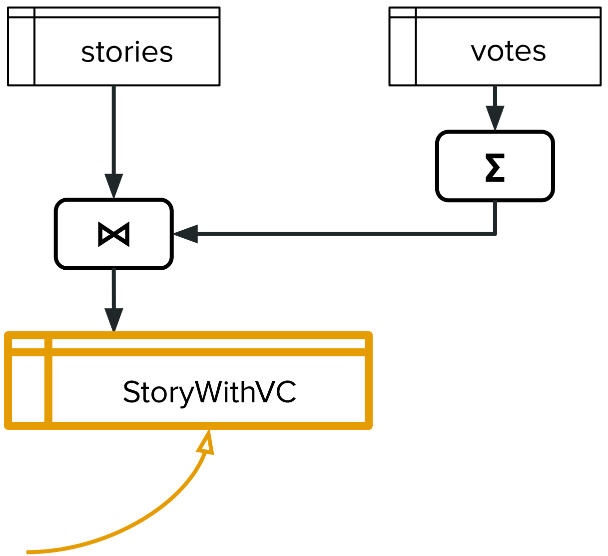
Partial State in Dataflow-Based Materialized Views

Learning to Forget

- Chances are that **most** entries in the view are not accessed.
  - Old and unpopular stories are **wasting memory**.
- Need to **evict** old entries, and only add new ones **on demand**.
- Three main contributions:
  - Notion of *missing state* in materialized views.
  - *Upqueries* to populate missing state using dataflow.
  - Implementation and evaluation of partial state in Noria.

View and Query are Separate

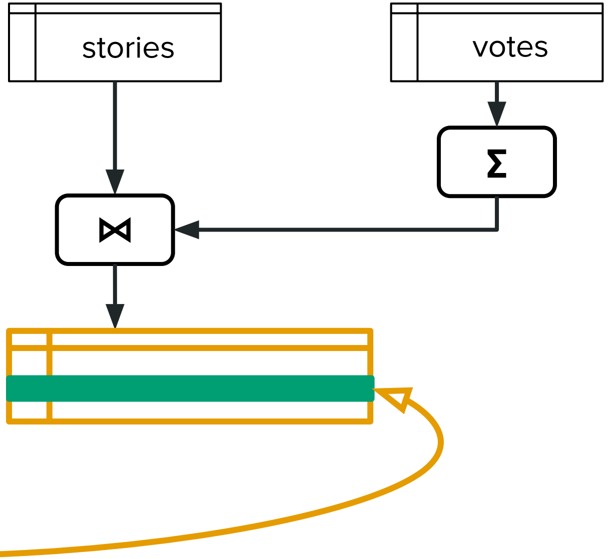
```
CREATE MATERIALIZED VIEW
 StoryWithVC
AS SELECT
 stories.*,
 COUNT(votes.user) AS votes
FROM stories
JOIN votes
 ON (votes.story_id = stories.id)
GROUP BY stories.id;
```




```
SELECT * FROM StoryWithVC WHERE id = ?
```

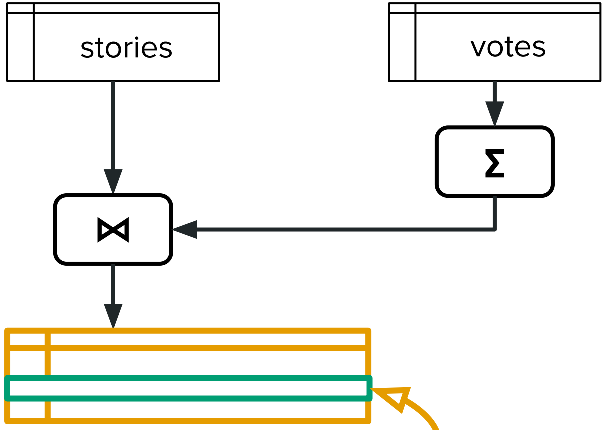
View Must Know Query Parameter(s)

```
CREATE MATERIALIZED VIEW
 StoryWithVC
AS SELECT
 stories.*,
 COUNT(votes.user) AS votes
FROM stories
JOIN votes
 ON (votes.story_id = stories.id)
GROUP BY stories.id
WHERE stories.id = ?;
```



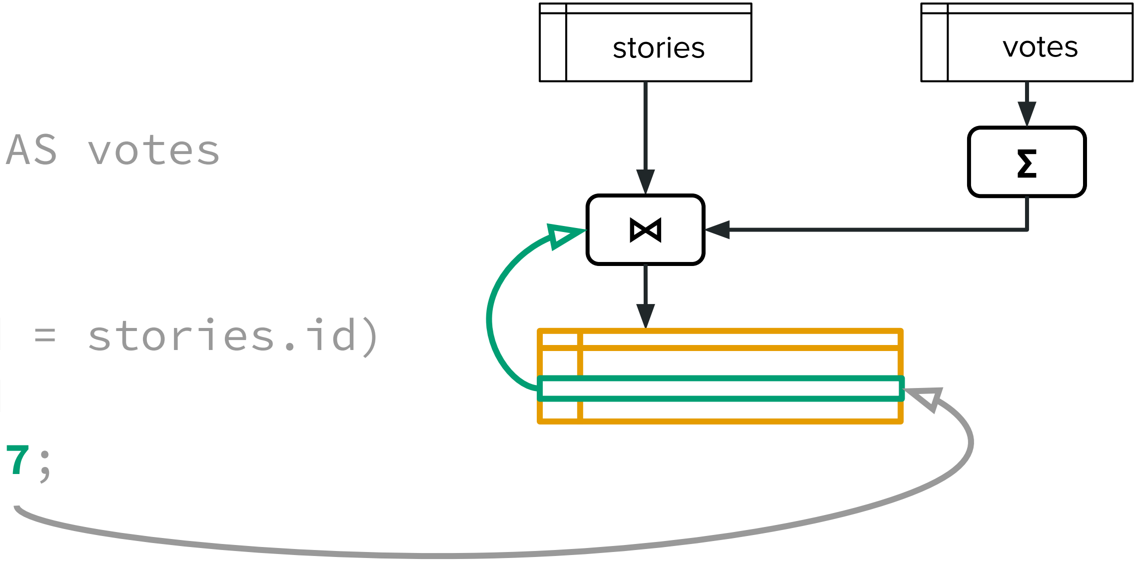
Queries Can Miss in Materialized View

```
CREATE MATERIALIZED VIEW
 StoryWithVC
AS SELECT
 stories.*,
 COUNT(votes.user) AS votes
FROM stories
JOIN votes
 ON (votes.story_id = stories.id)
GROUP BY stories.id
WHERE stories.id = 7;
```



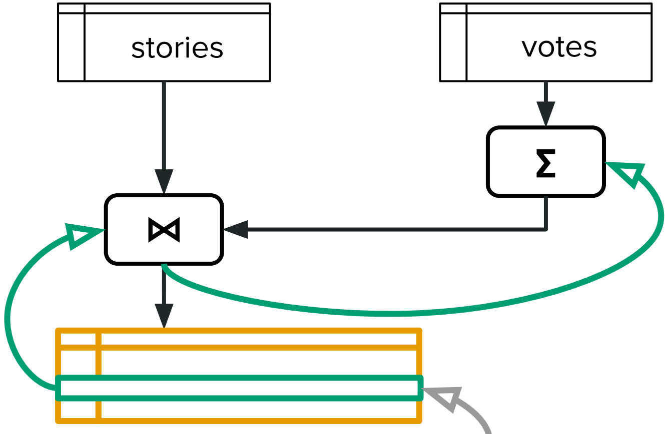
Misses Trigger Upqueries

```
CREATE MATERIALIZED VIEW
 StoryWithVC
AS SELECT
 stories.*,
 COUNT(votes.user) AS votes
FROM stories
JOIN votes
 ON (votes.story_id = stories.id)
GROUP BY stories.id
WHERE stories.id = 7;
```



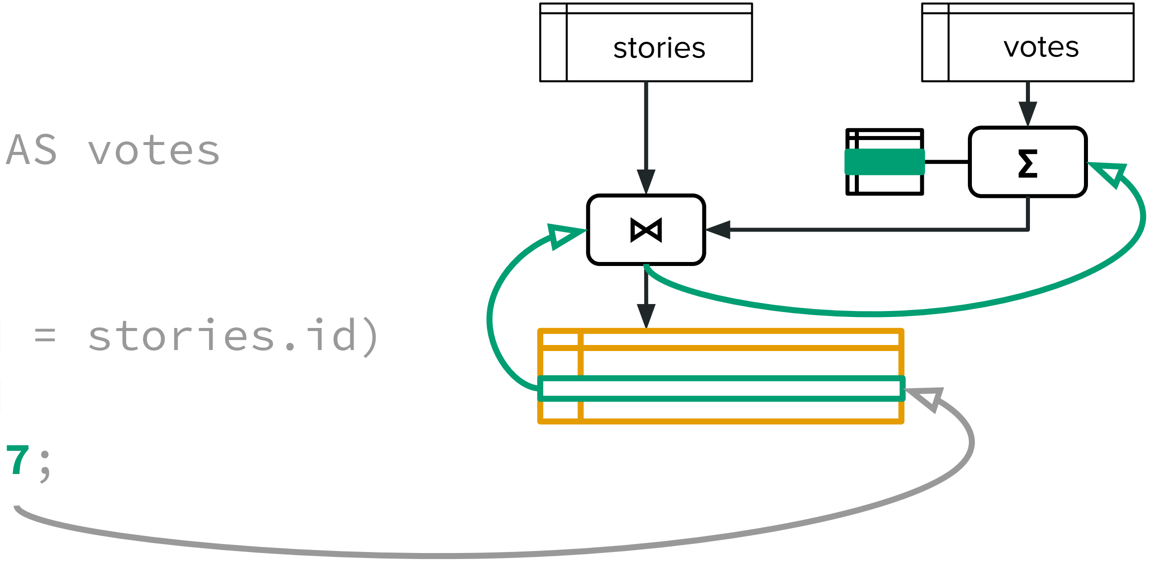
Upqueries Can Trigger Further Upqueries

```
CREATE MATERIALIZED VIEW
 StoryWithVC
AS SELECT
 stories.*,
 COUNT(votes.user) AS votes
FROM stories
JOIN votes
 ON (votes.story_id = stories.id)
GROUP BY stories.id
WHERE stories.id = 7;
```



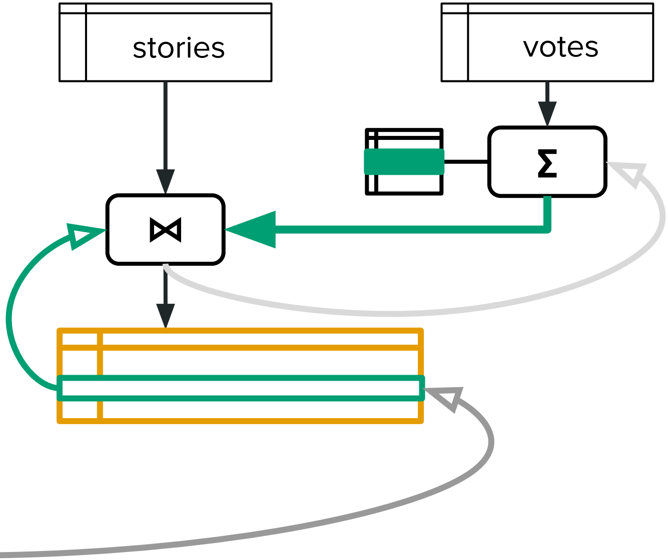
Answer May Reside in Intermediate State

```
CREATE MATERIALIZED VIEW
 StoryWithVC
AS SELECT
 stories.*,
 COUNT(votes.user) AS votes
FROM stories
JOIN votes
 ON (votes.story_id = stories.id)
GROUP BY stories.id
WHERE stories.id = 7;
```



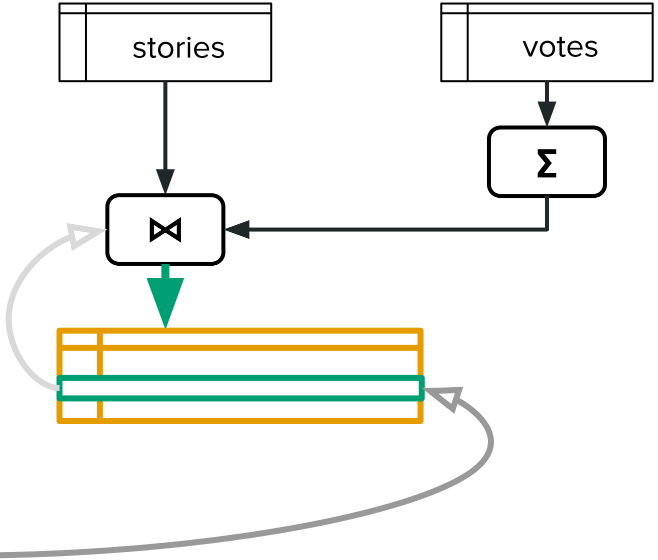
Response Uses Normal Dataflow

```
CREATE MATERIALIZED VIEW
 StoryWithVC
AS SELECT
 stories.*,
 COUNT(votes.user) AS votes
FROM stories
JOIN votes
 ON (votes.story_id = stories.id)
GROUP BY stories.id
WHERE stories.id = 7;
```



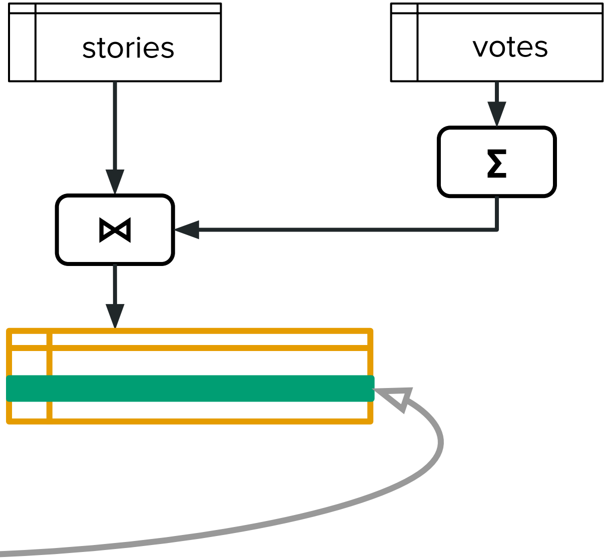
Response Uses Normal Dataflow

```
CREATE MATERIALIZED VIEW
 StoryWithVC
AS SELECT
 stories.*,
 COUNT(votes.user) AS votes
FROM stories
JOIN votes
 ON (votes.story_id = stories.id)
GROUP BY stories.id
WHERE stories.id = 7;
```



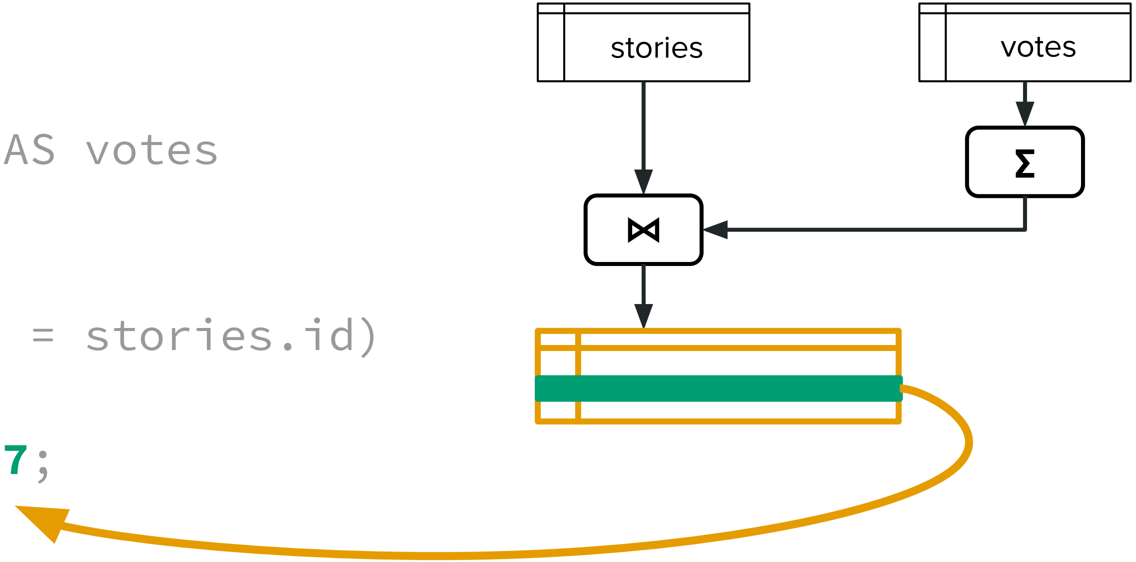
Response Uses Normal Dataflow

```
CREATE MATERIALIZED VIEW
 StoryWithVC
AS SELECT
 stories.*,
 COUNT(votes.user) AS votes
FROM stories
JOIN votes
 ON (votes.story_id = stories.id)
GROUP BY stories.id
WHERE stories.id = 7;
```



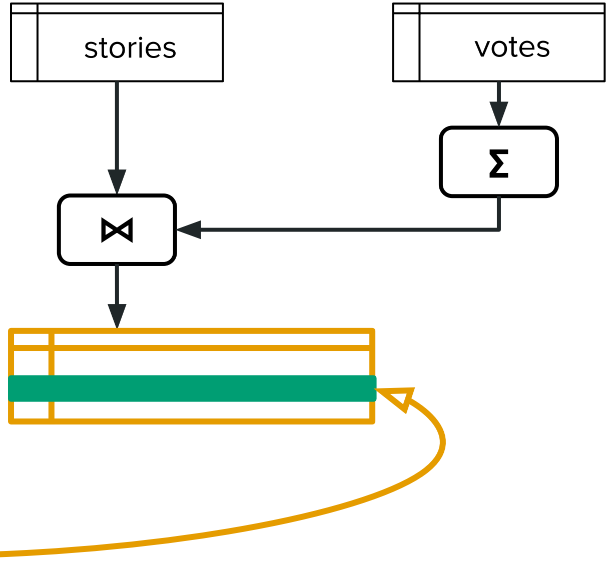
Response Uses Normal Dataflow

```
CREATE MATERIALIZED VIEW
 StoryWithVC
AS SELECT
 stories.*,
 COUNT(votes.user) AS votes
FROM stories
JOIN votes
 ON (votes.story_id = stories.id)
GROUP BY stories.id
WHERE stories.id = 7;
```



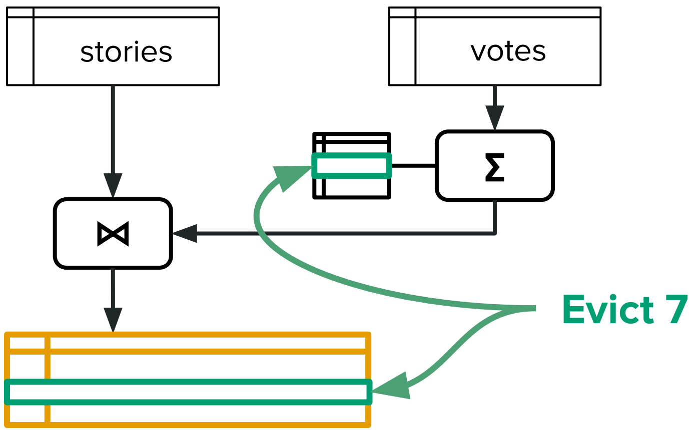
Next Query with Same Parameter is Fast

```
CREATE MATERIALIZED VIEW
 StoryWithVC
AS SELECT
 stories.*,
 COUNT(votes.user) AS votes
FROM stories
JOIN votes
 ON (votes.story_id = stories.id)
GROUP BY stories.id
WHERE stories.id = 7;
```



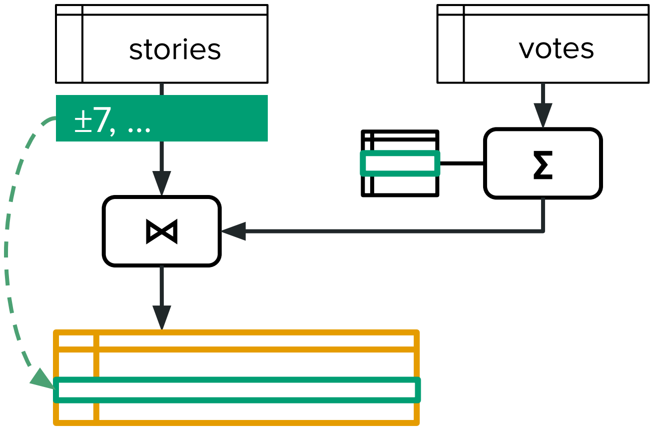
To Evict: Mark as Missing Again

```
CREATE MATERIALIZED VIEW
 StoryWithVC
AS SELECT
 stories.*,
 COUNT(votes.user) AS votes
FROM stories
JOIN votes
 ON (votes.story_id = stories.id)
GROUP BY stories.id
WHERE stories.id = ?;
```



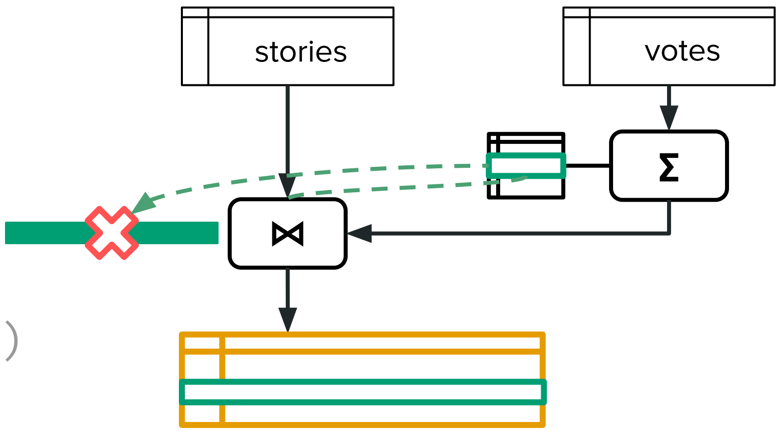
No Need to Update Missing State!

```
CREATE MATERIALIZED VIEW
 StoryWithVC
AS SELECT
 stories.*,
 COUNT(votes.user) AS votes
FROM stories
JOIN votes
 ON (votes.story_id = stories.id)
GROUP BY stories.id
WHERE stories.id = ?;
```



No Need to Update Missing State!

```
CREATE MATERIALIZED VIEW
 StoryWithVC
AS SELECT
 stories.*,
 COUNT(votes.user) AS votes
FROM stories
JOIN votes
 ON (votes.story_id = stories.id)
GROUP BY stories.id
WHERE stories.id = ?;
```



Intermission

Related work

Materialized View Maintenance

- Primarily targets analytics workloads ➜ infrequent reads.
- Little or no support for on-demand queries.
- No support for eviction.

Automated Caching Systems

- Few are general-purpose.
- Many only support invalidation, not updates.
- Often limited to specific database interaction, not general SQL.

Dataflow and Stream Processing

- Usually focused on write performance.
- Focus on strong consistency at the cost of read latency.
- Limited support for on-demand compute & eviction.

Are we done?

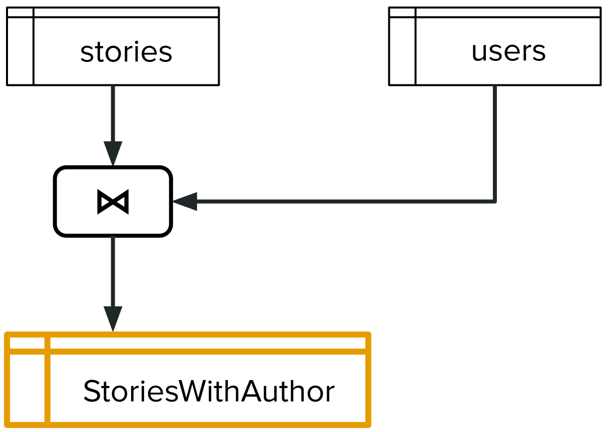
In Practice, Things are Hard

- Must ensure that data changes take effect exactly once.
- Traditionally easy, but hard in this model because:
  - Upqueries hold past state which may be concurrently updated.
  - Updates may be discarded early.
- Many hazards (see thesis), but we'll focus on one.

Incongruent Join Evictions

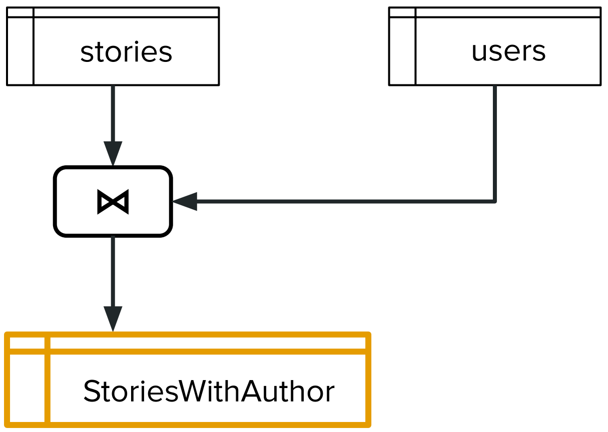
What is an Incongruent Join?

```
CREATE MATERIALIZED VIEW
 StoriesWithAuthor
AS SELECT
 stories.*,
 users.name AS aname,
FROM stories
JOIN users
 ON (stories.author = users.id)
WHERE stories.id = ?;
```



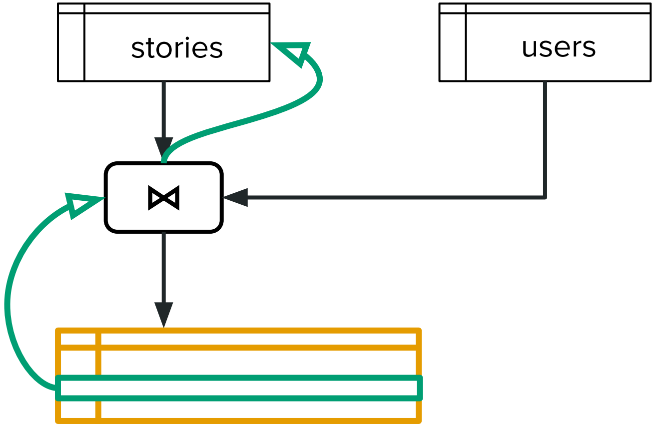
Query Key ≠ Join Key

```
CREATE MATERIALIZED VIEW
 StoriesWithAuthor
AS SELECT
 stories.*,
 users.name AS aname,
FROM stories
JOIN users
 ON (stories.author = users.id)
WHERE stories.id = ?;
```



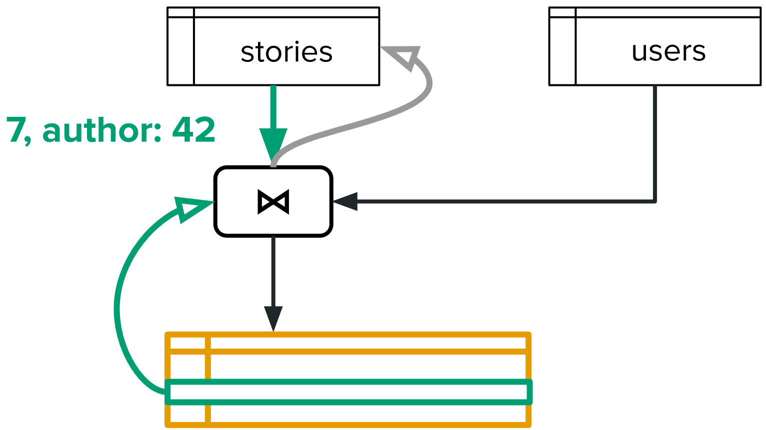
Upquery Works Correctly

```
CREATE MATERIALIZED VIEW
 StoriesWithAuthor
AS SELECT
 stories.*,
 users.name AS aname,
FROM stories
JOIN users
 ON (stories.author = users.id)
WHERE stories.id = 7;
```



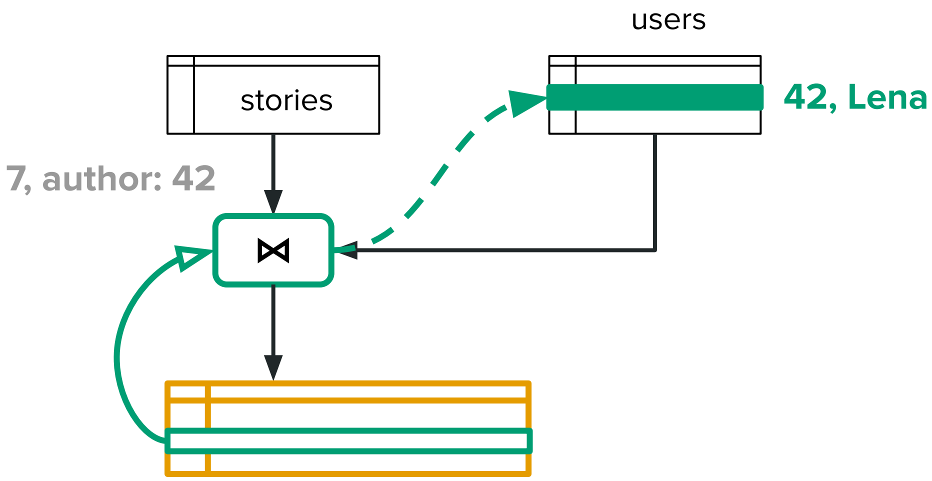
Upquery Works Correctly

```
CREATE MATERIALIZED VIEW
 StoriesWithAuthor
AS SELECT
 stories.*,
 users.name AS aname,
FROM stories
JOIN users
 ON (stories.author = users.id)
WHERE stories.id = 7;
```



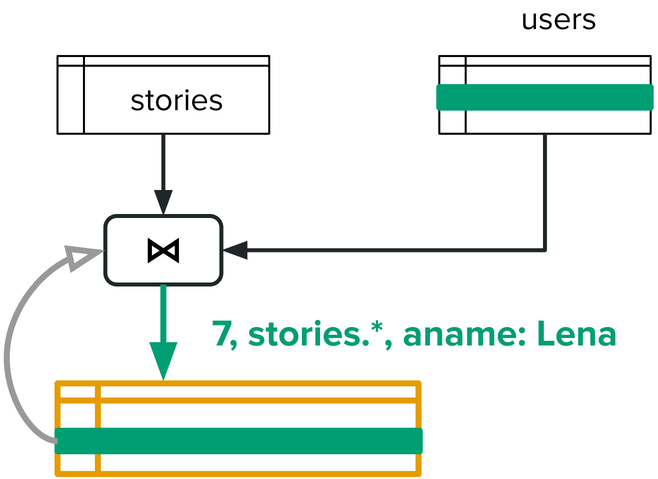
Upquery Works Correctly

```
CREATE MATERIALIZED VIEW
 StoriesWithAuthor
AS SELECT
 stories.*,
 users.name AS aname,
FROM stories
JOIN users
 ON (stories.author = users.id)
WHERE stories.id = 7;
```

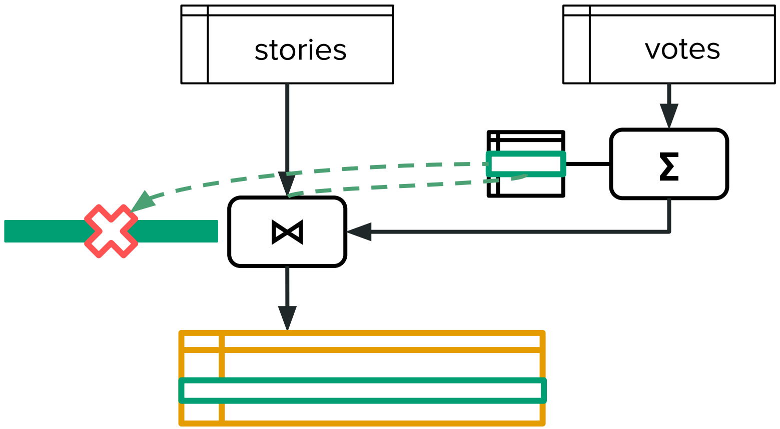


Upquery Works Correctly

```
CREATE MATERIALIZED VIEW
 StoriesWithAuthor
AS SELECT
 stories.*,
 users.name AS aname,
FROM stories
JOIN users
 ON (stories.author = users.id)
WHERE stories.id = 7;
```

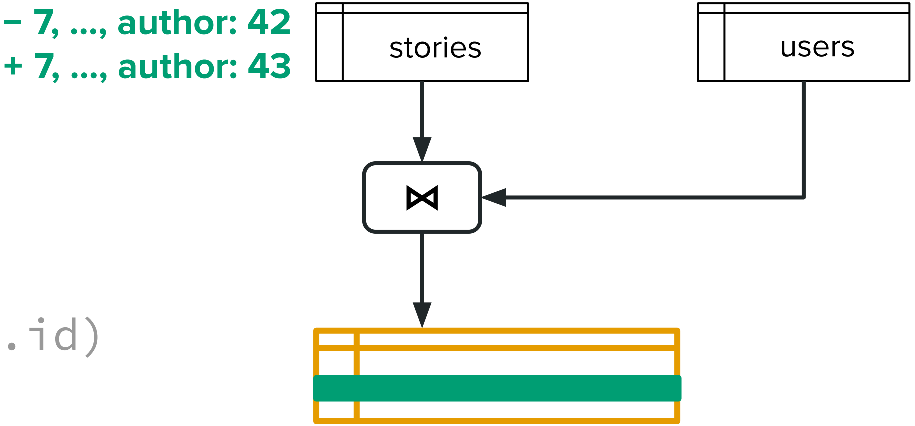


Recall This Figure?



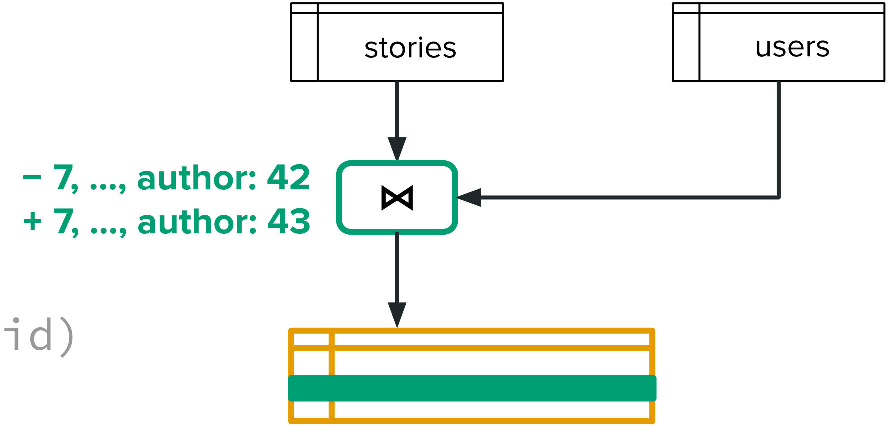
What if the Author Changes?

```
CREATE MATERIALIZED VIEW
 StoriesWithAuthor
AS SELECT
 stories.*,
 users.name AS aname,
FROM stories
JOIN users
 ON (stories.author = users.id)
WHERE stories.id = 7;
```



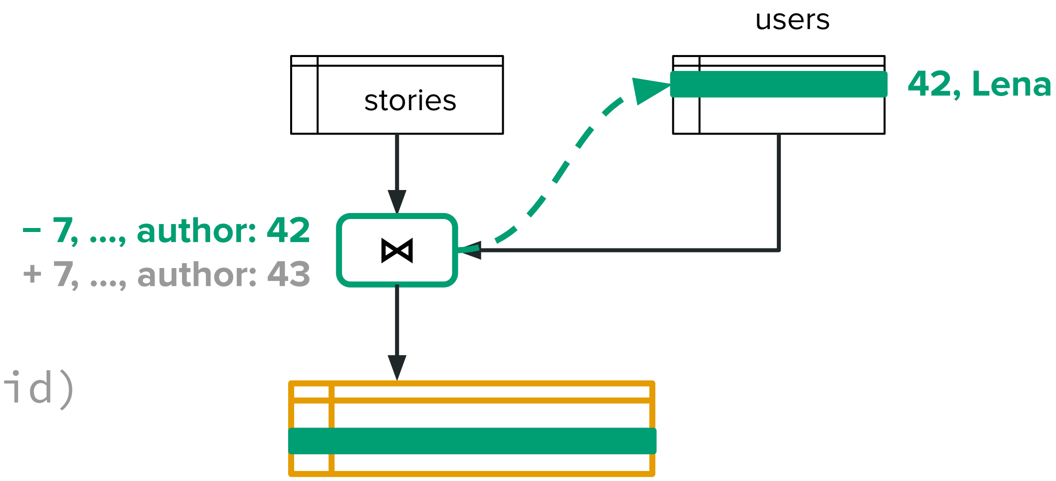
Change Must Propagate to the View

```
CREATE MATERIALIZED VIEW
 StoriesWithAuthor
AS SELECT
 stories.*,
 users.name AS aname,
FROM stories
JOIN users
 ON (stories.author = users.id)
WHERE stories.id = 7;
```



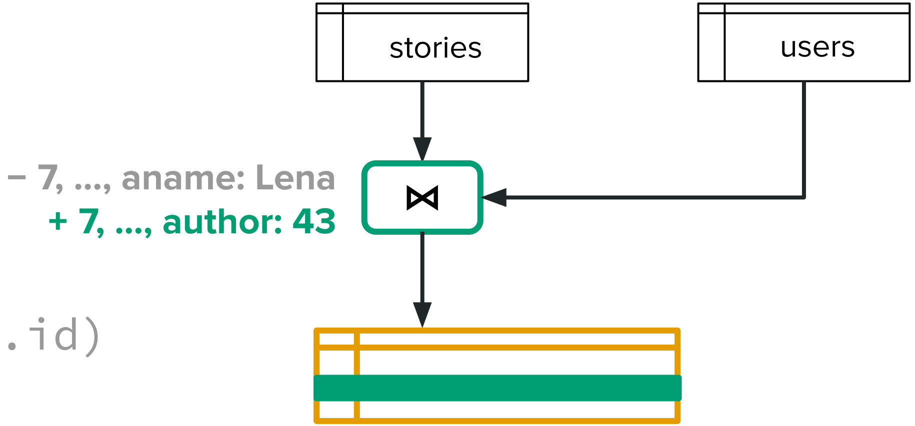
Each Change is Joined

```
CREATE MATERIALIZED VIEW
 StoriesWithAuthor
AS SELECT
 stories.*,
 users.name AS aname,
FROM stories
JOIN users
 ON (stories.author = users.id)
WHERE stories.id = 7;
```



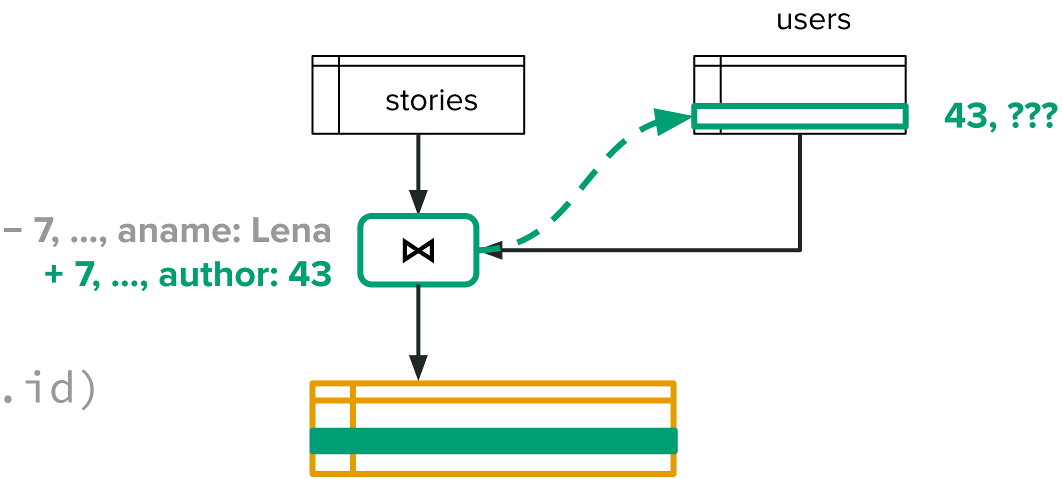
Just One More Step

```
CREATE MATERIALIZED VIEW
 StoriesWithAuthor
AS SELECT
 stories.*,
 users.name AS aname,
FROM stories
JOIN users
 ON (stories.author = users.id)
WHERE stories.id = 7;
```



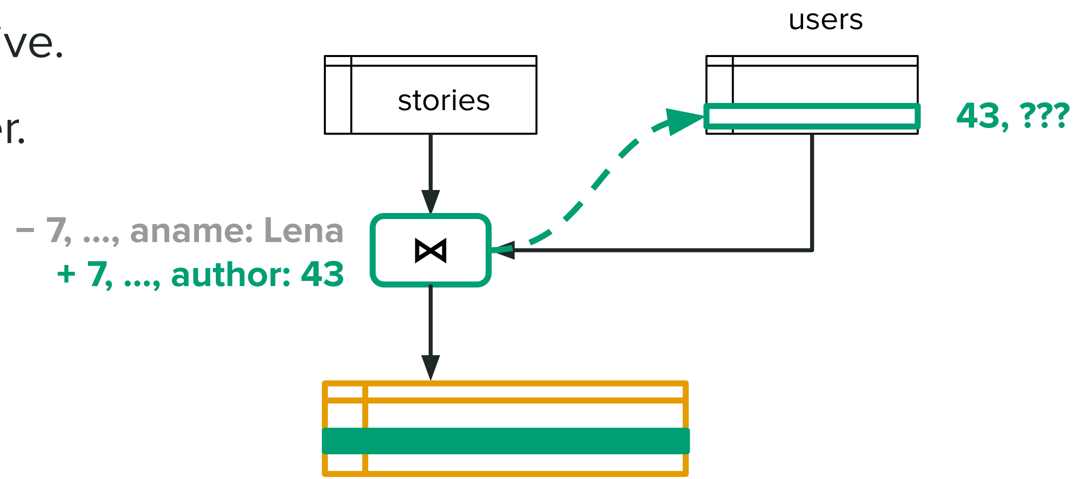
State for New Author is Missing!

```
CREATE MATERIALIZED VIEW
 StoriesWithAuthor
AS SELECT
 stories.*,
 users.name AS aname,
FROM stories
JOIN users
 ON (stories.author = users.id)
WHERE stories.id = 7;
```



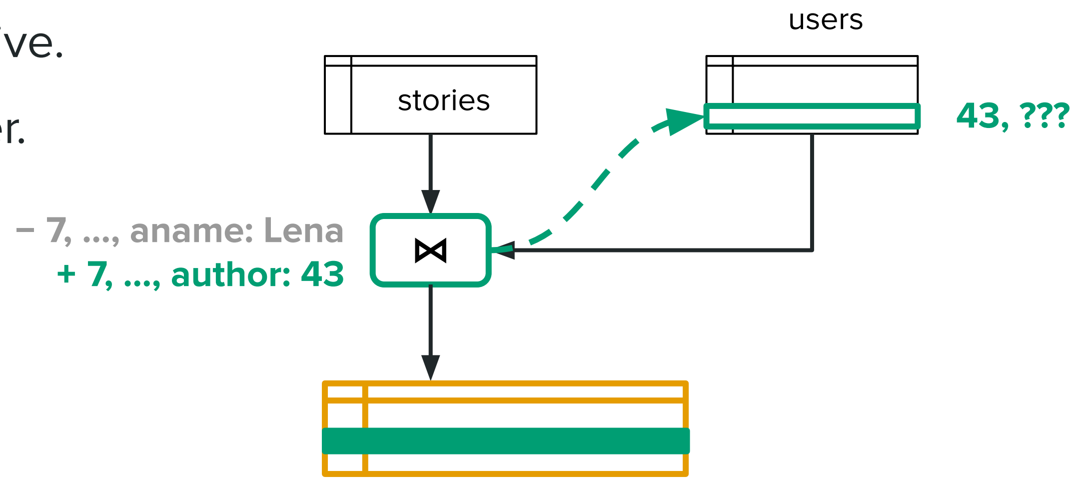
What Do We Do?

- Cannot produce needed update!
- Cannot forward just the negative.
- Cannot drop update altogether.



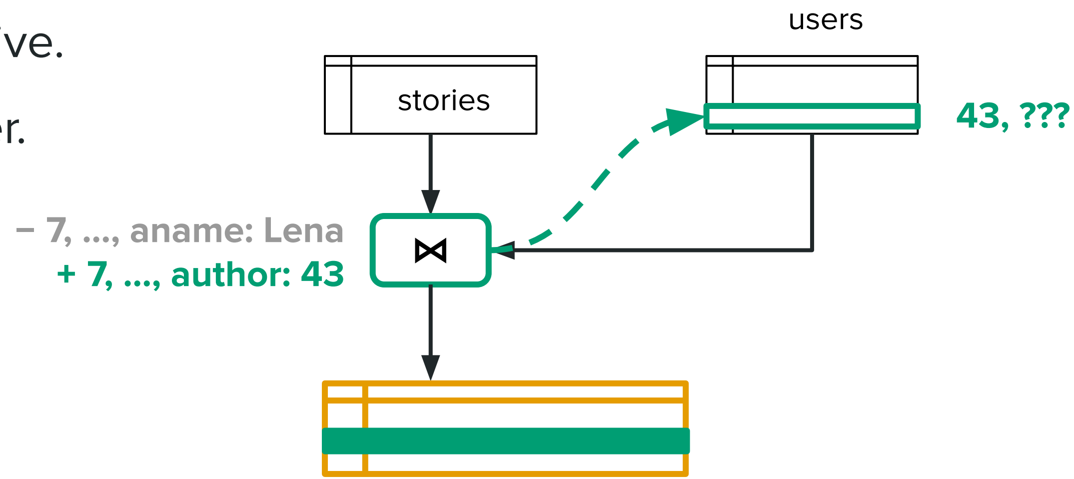
What Do We Do?

- Cannot produce needed update!
- Cannot forward just the negative.
- Cannot drop update altogether.
- Fill missing state?



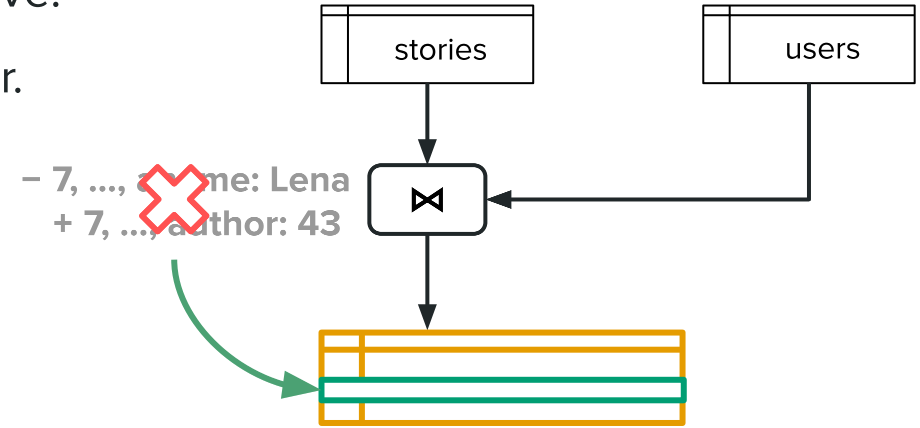
What Do We Do?

- Cannot produce needed update!
- Cannot forward just the negative.
- Cannot drop update altogether.
- ~~Fill missing state?~~



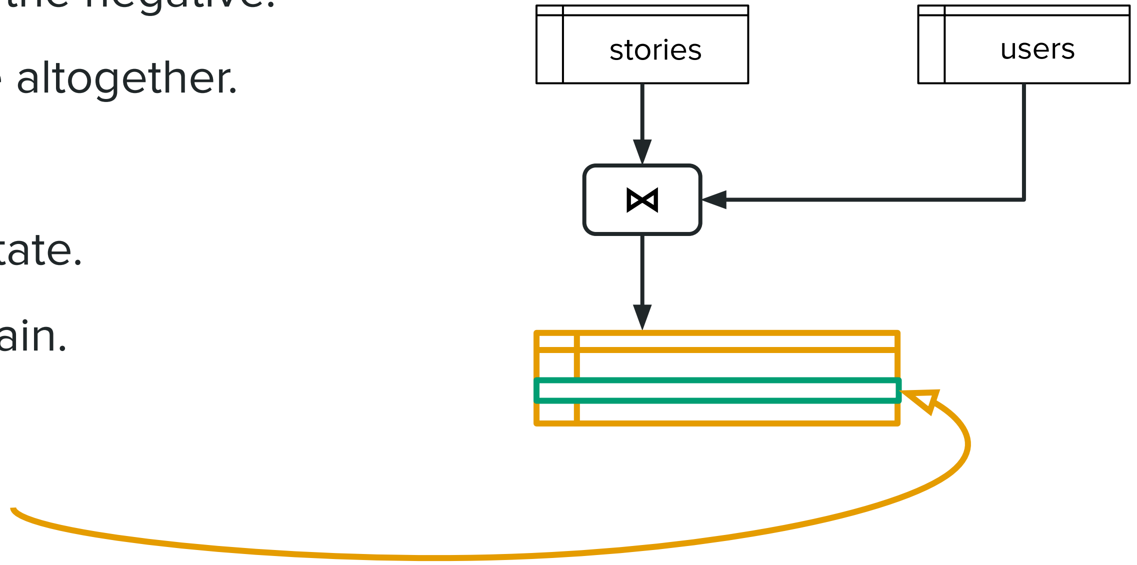
What Do We Do?

- Cannot produce needed update!
- Cannot forward just the negative.
- Cannot drop update altogether.
- ~~Fill missing state?~~
- **Evict** downstream state.



What Do We Do?

- Cannot produce needed update!
- Cannot forward just the negative.
- Cannot drop update altogether.
- ~~Fill missing state?~~
- **Evict** downstream state.
- Next query fills it again.



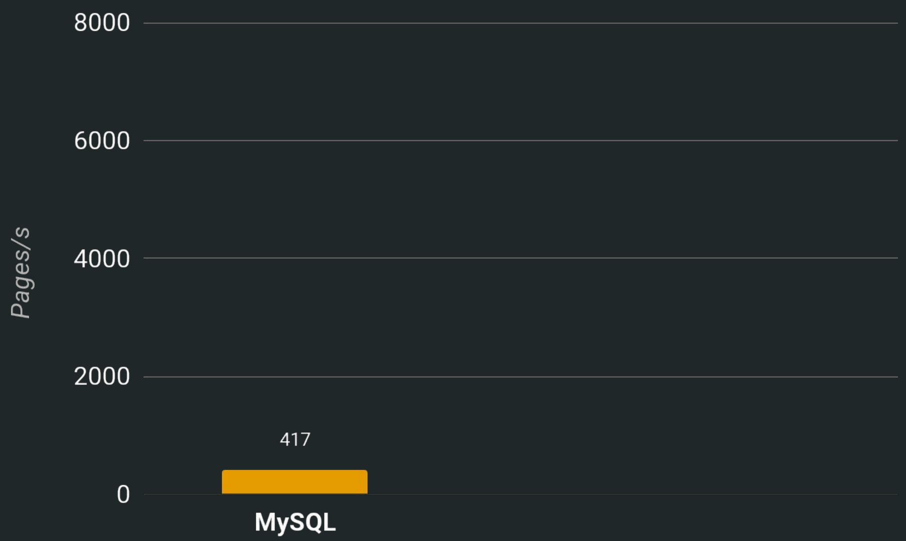
Does it work?

Need a Realistic Test Subject

- Lobste.rs: a Hacker News-like news aggregator.
  - Users submit stories, vote for and comment on them, etc.
  - Open-source, so we can see the queries.
  - Data statistics available, so we know the workload.
- Workload generator: synthesize Lobste.rs-like requests.

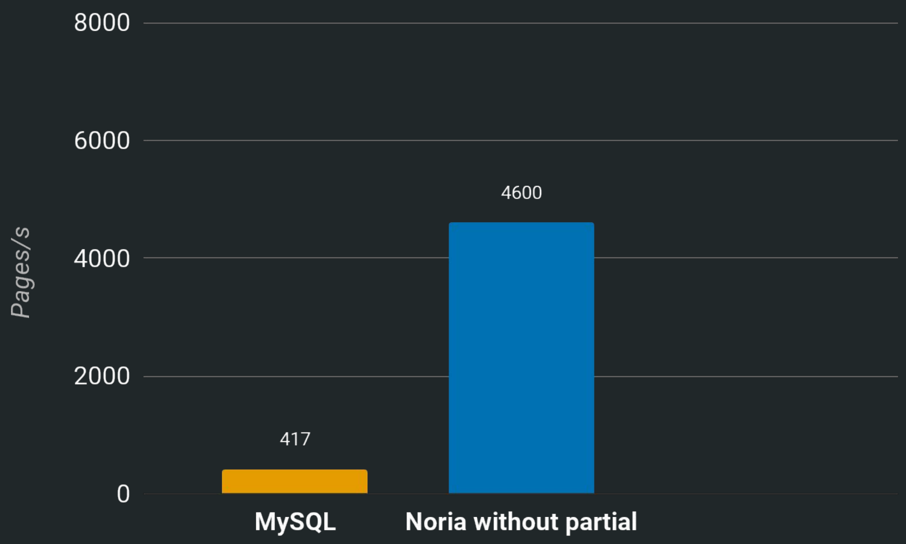
Throughput

Fixed available resources.



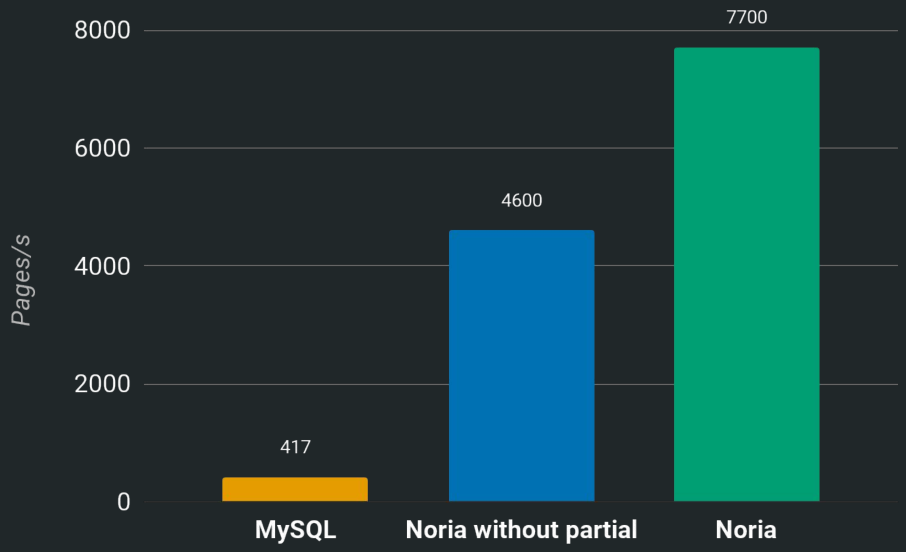
Throughput

Fixed available resources.



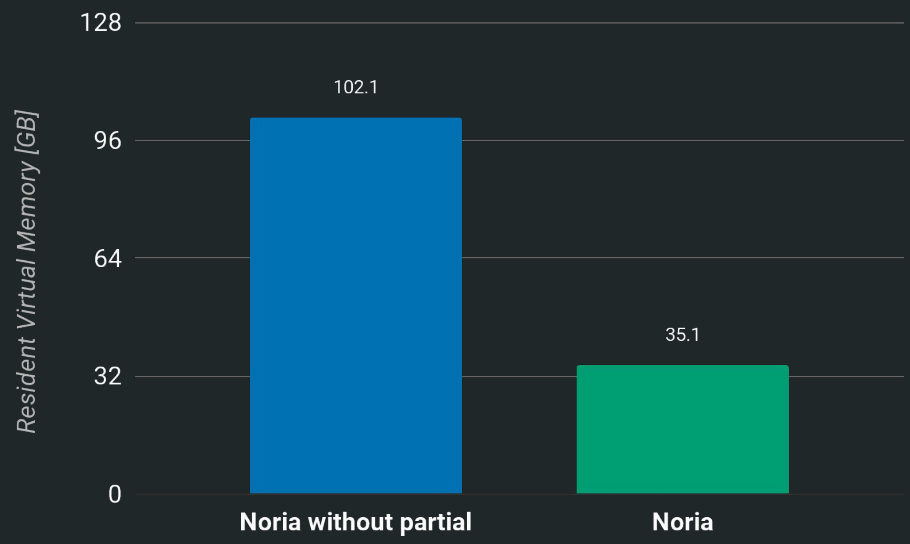
Throughput

Fixed available resources.



Memory use

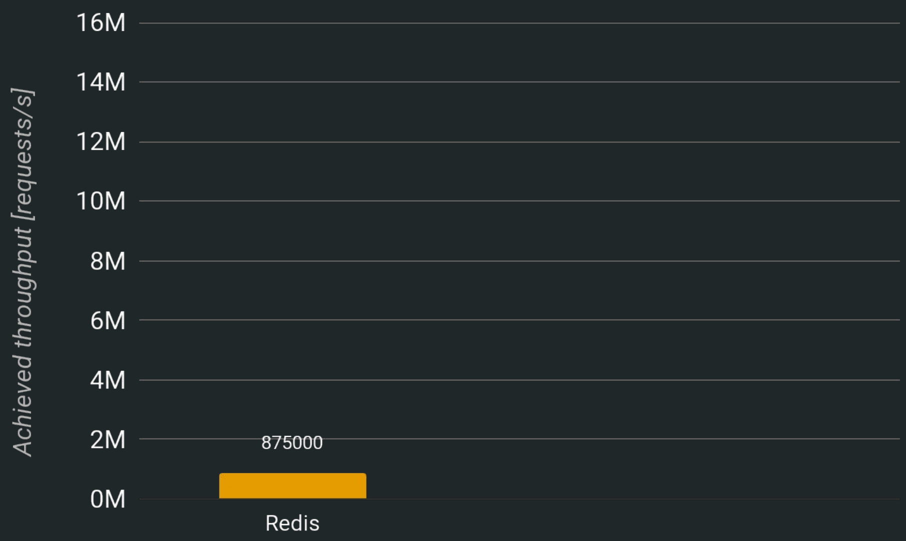
Fixed throughput & runtime.



Noria vs. cache

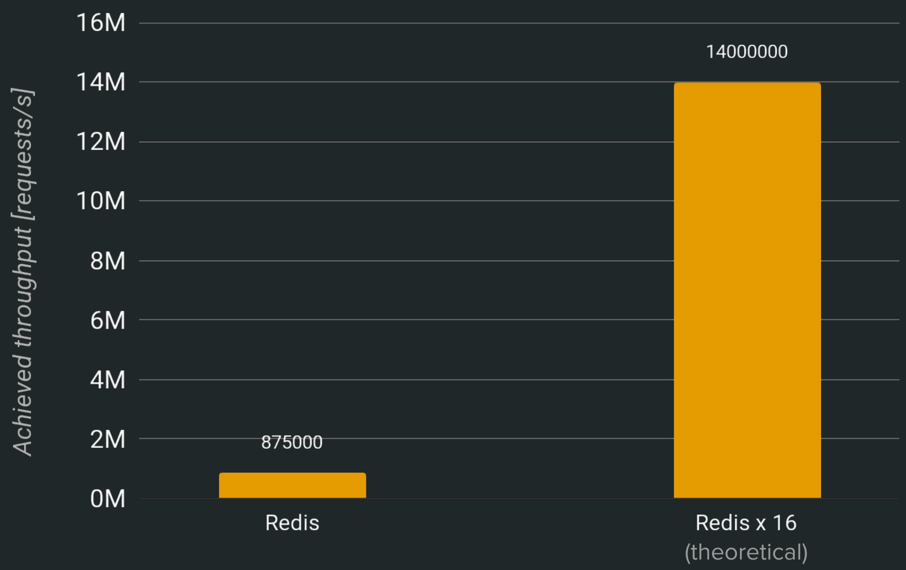
vs. Redis

Idealized cache workload.



vs. Redis

Idealized cache workload.

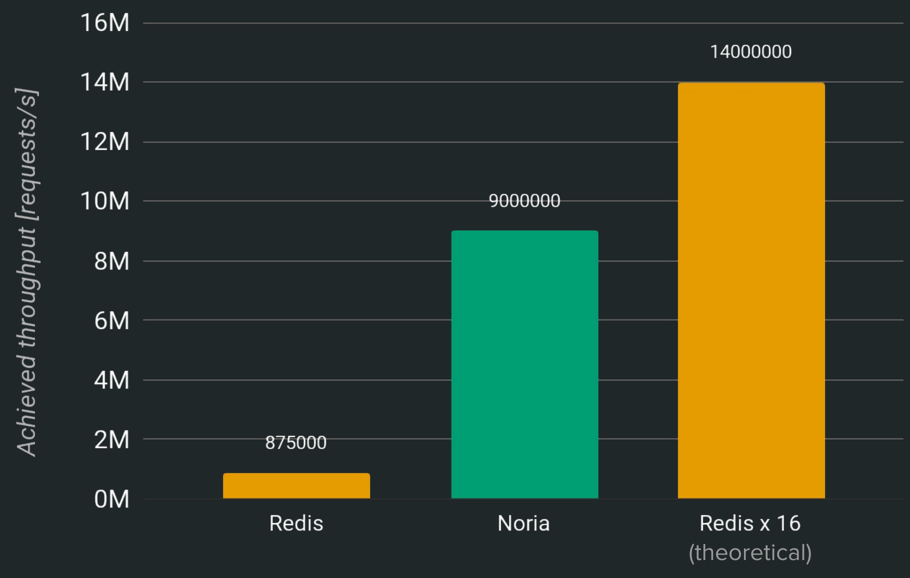


Redis is single-threaded, so 16x is extrapolated.

vs. Redis

Idealized cache workload.

Redis is single-threaded, so 16x is extrapolated.



Wrapping things up

Future work

Noria is neither perfect nor complete.

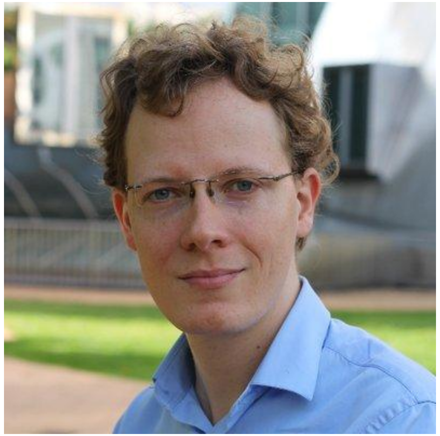
- Range queries, cursors, time-windowed operators.
- Upstream database integration.
- Maintaining downstream views.
- Fault tolerance.

Acknowledgements

Robert Morris

M. Frans Kaashoek

Sam Madden



Malte Schwarzkopf

Parallel & Distributed Operating Systems Group

Conclusion

My thesis enables **materialized views** to be used as **caches**.

It does so by allowing state to be **missing** from materializations,

and using **upqueries** to populate missing state on demand.

The resulting system provides **automated** caching for SQL queries,

and reduces the need for complex, ad hoc caching logic.

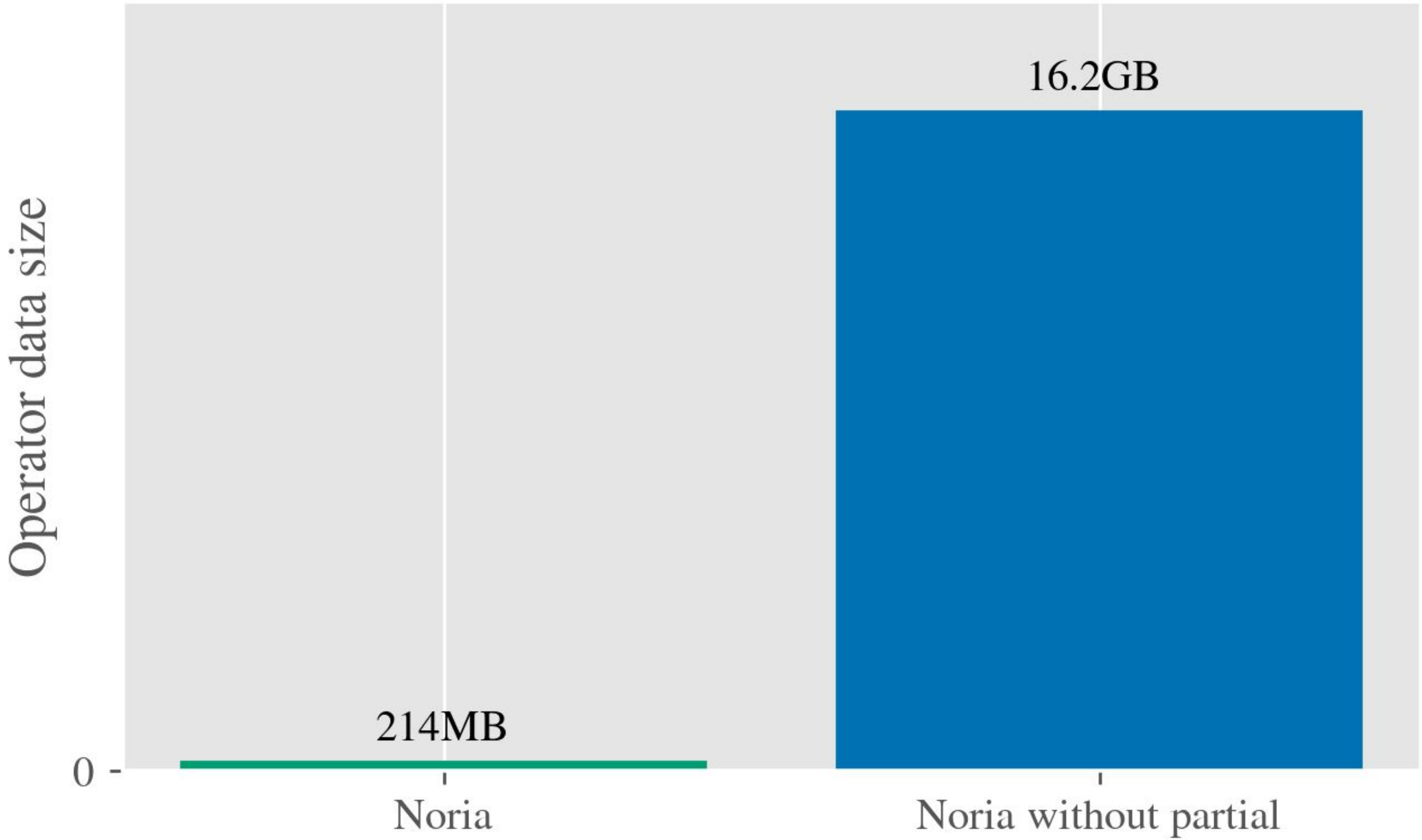
Thank you — please ask questions!

jon@thesquareplanet.com

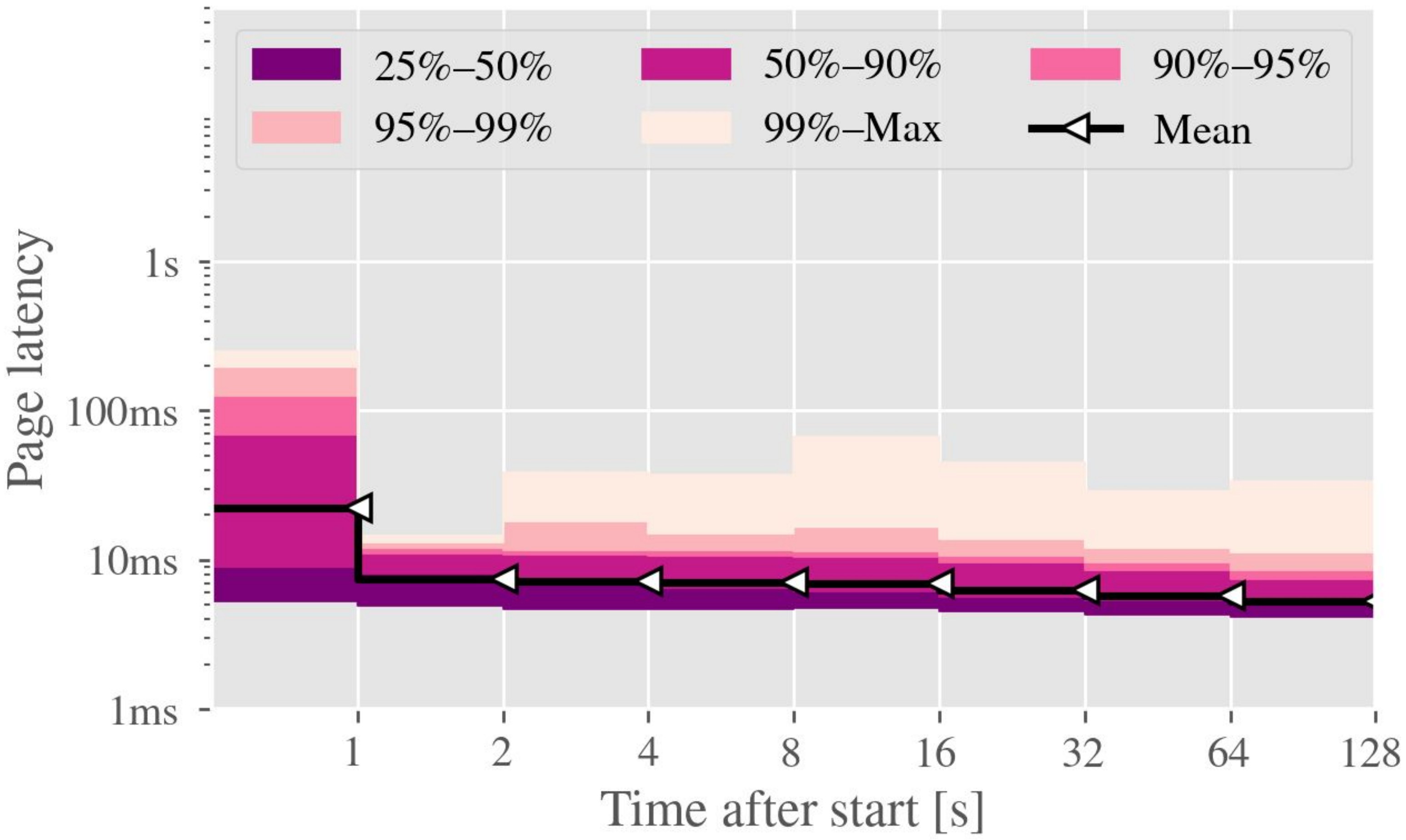
Backup slides

Page	%	W	Q	Description
Story	55.8	1	14	Renders an individual story's page, including its popularity score, comments, and the scores of its comments.
Front page	30.1	0	14	Lists the 25 most highly scored stories, along with their authors and scores.
User	6.7	0	7	Renders a user summary page, including what story "tags" they contribute to.
Comments	4.7	0	9	Like the front page, but for comments.
Recent	1.0	0	14	25 most recently added stories, along with their authors and scores.
Vote	1.2	1	2	Vote up/down a given comment or story.
Comment	0.4	2	5	Add a new comment to a story.

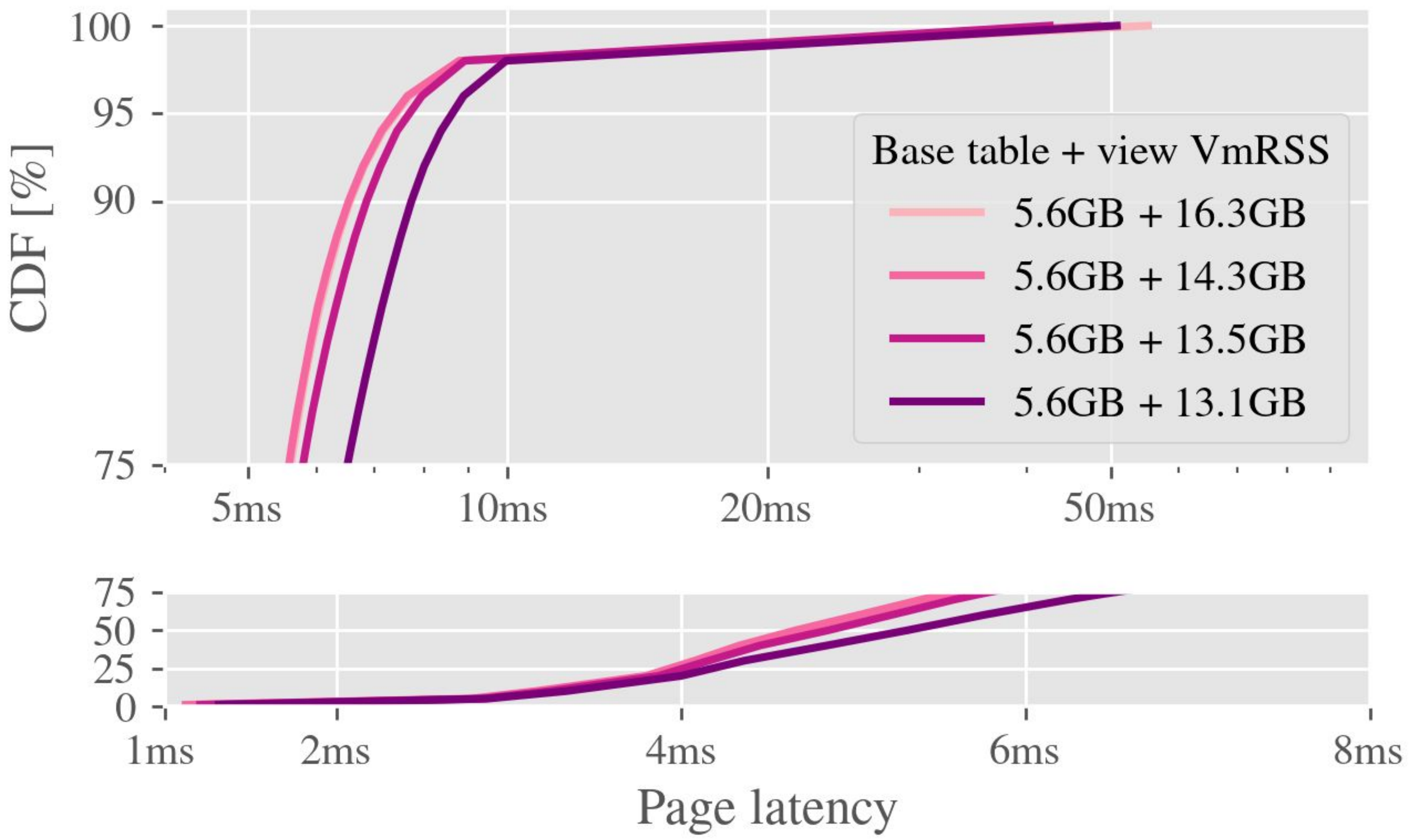
Table 6.1.: Pages in Lobsters. % indicates the percentage of requests that load the given page. W is the number of writes performed by a given page. Q is the number of (read) queries a page issues.

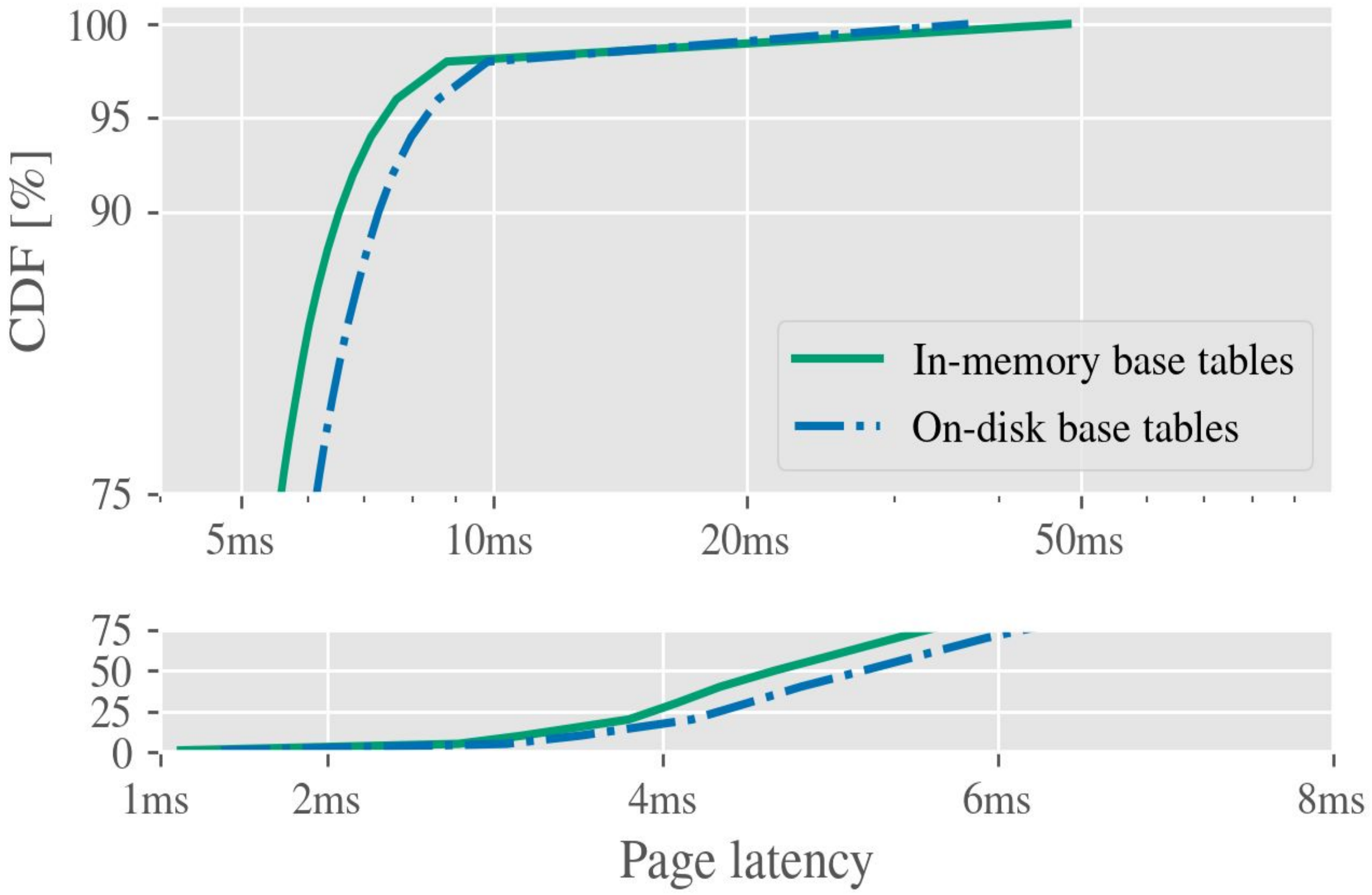


16.2GB
214MB
Operator data size
0
Noria
Noria without partial

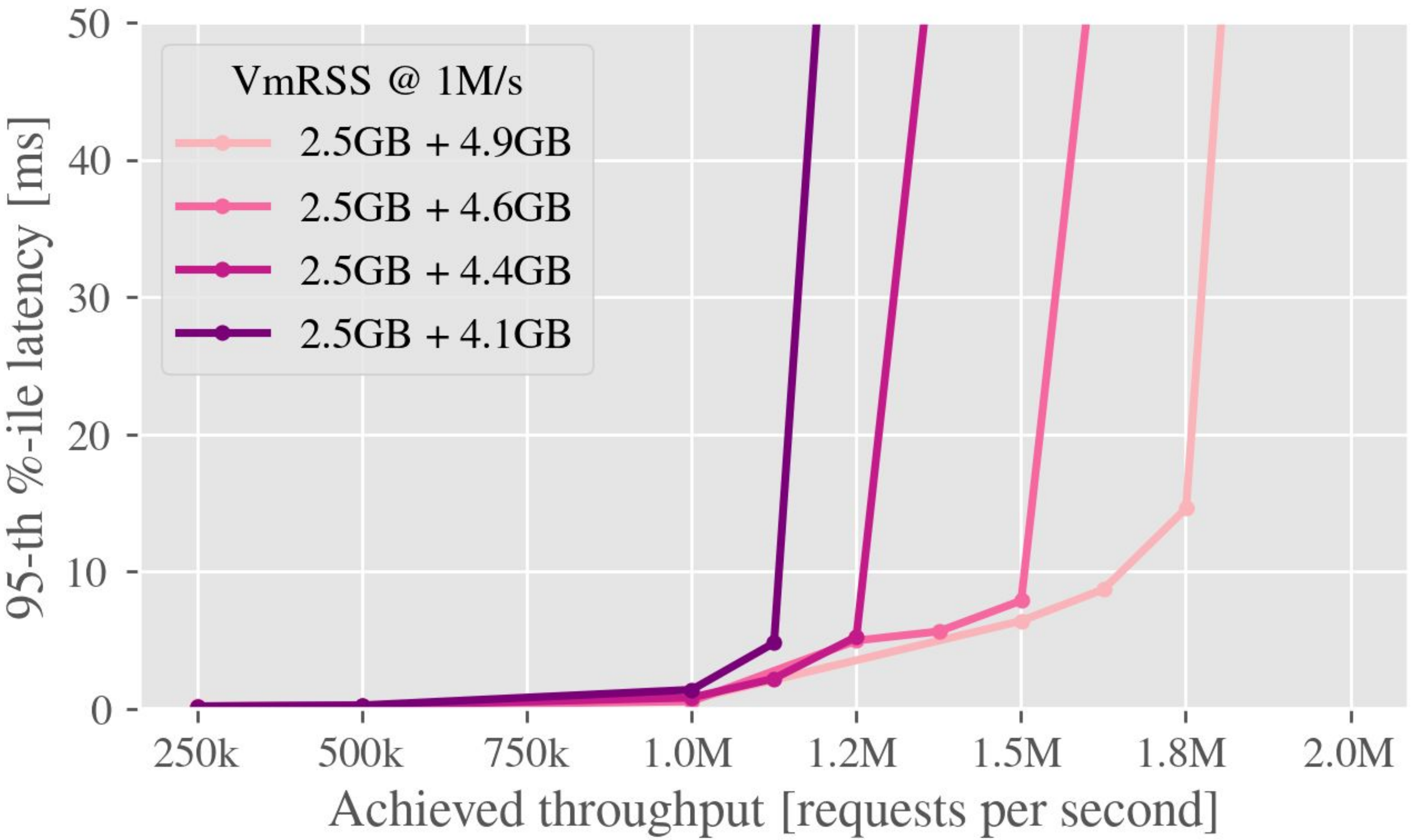


25%–50%
50%–90%
90%–95%
95%–99%
99%–Max
Mean
Page latency
1s
100ms
10ms
1ms
1
2
4
8
16
32
64
128
Time after start [s]

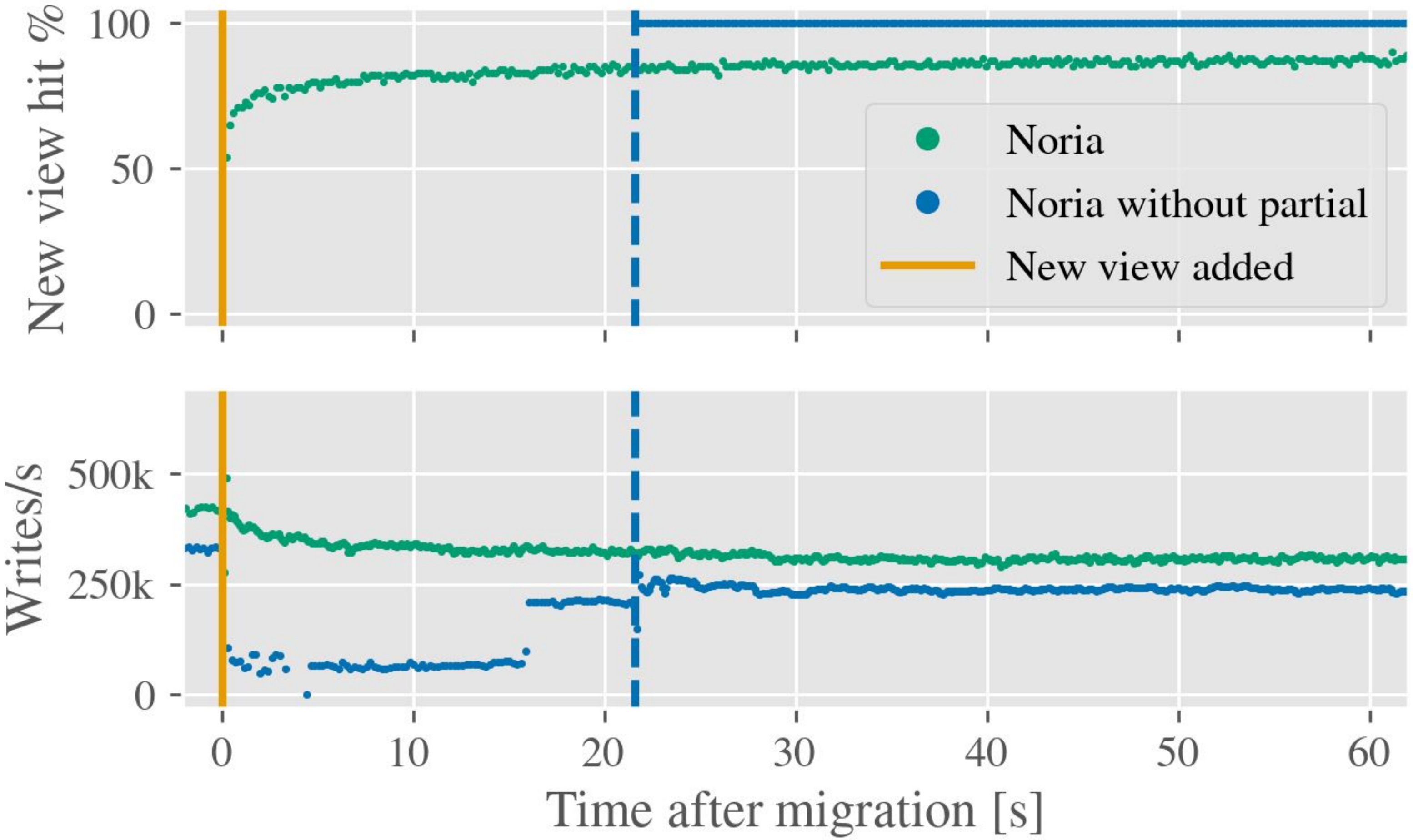

100
95
90
75
CDF [%]
5ms
10ms
20ms
50ms
Base table + view VmRSS
5.6GB + 16.3GB
5.6GB + 14.3GB
5.6GB + 13.5GB
5.6GB + 13.1GB
75
50
25
0
1ms
2ms
4ms
6ms
8ms
Page latency



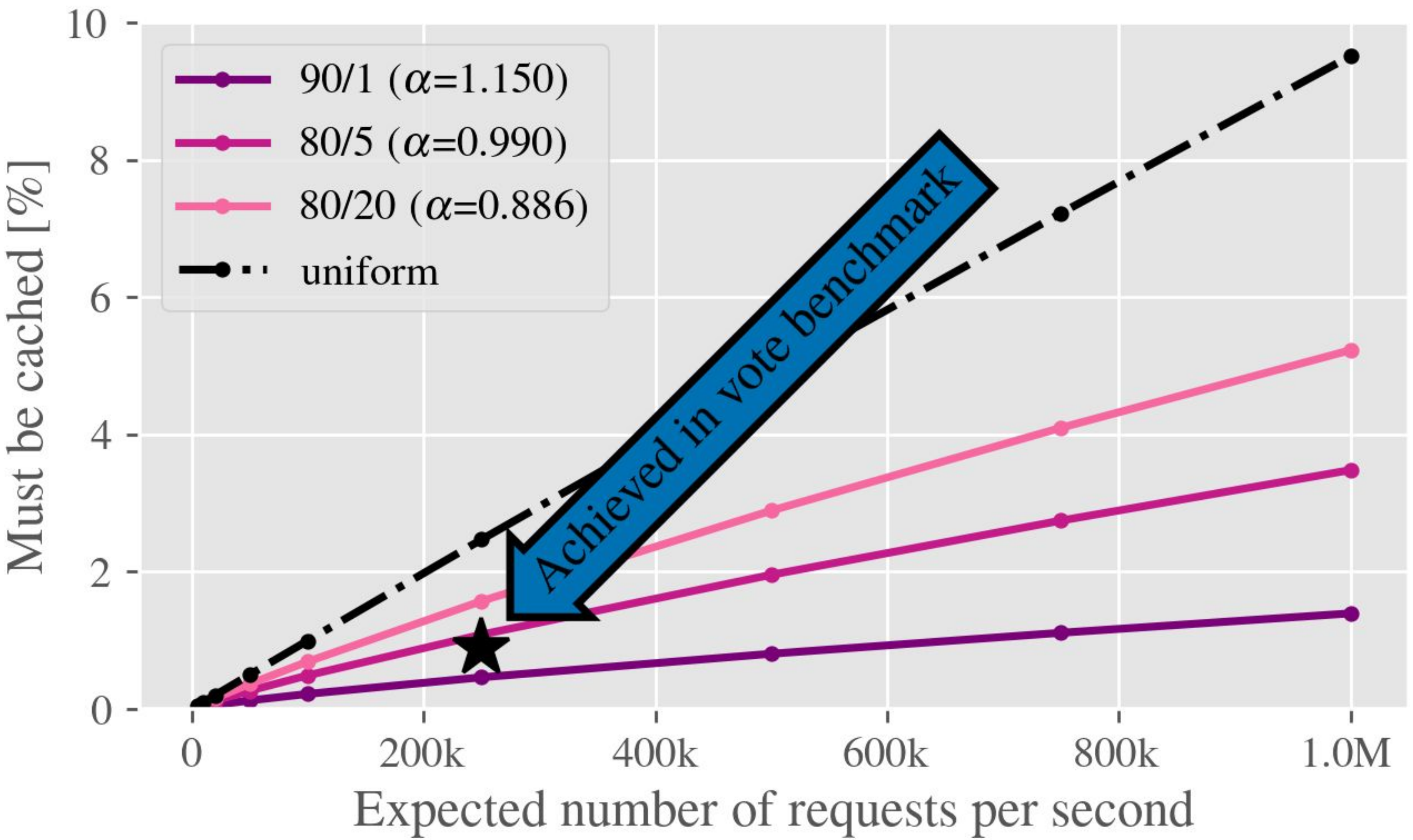
100
95
90
75
CDF [%]
In-memory base tables
On-disk base tables
5ms
10ms
20ms
50ms
75
50
25
0
1ms
2ms
4ms
6ms
8ms
Page latency



VmRSS @ 1M/s
2.5GB + 4.9GB
2.5GB + 4.6GB
2.5GB + 4.4GB
2.5GB + 4.1GB
95-th %-ile latency [ms]
0
10
20
30
40
50
250k
500k
750k
1.0M
1.2M
1.5M
1.8M
2.0M
Achieved throughput [requests per second]



New view hit %
100
50
0
Noria
Noria without partial
New view added
Writes/s
500k
250k
0
0
10
20
30
40
50
60
Time after migration [s]


90/1 (α=1.150)
80/5 (α=0.990)
80/20 (α=0.886)
uniform
Achieved in vote benchmark
Must be cached [%]
0
2
4
6
8
10
0
200k
400k
600k
800k
1.0M
Expected number of requests per second